

## 版权注意事项：

- 1、书籍版权归作者和出版社所有
- 2、本PDF仅限用于个人获取知识，进行私底下的知识交流
- 3、PDF获得者不得在互联网上以任何目的进行传播
- 4、如觉得书籍内容很赞，请购买正版实体书，支持作者
- 5、请于下载PDF后24小时内删除本PDF。



超级账本执行董事Brian Behlendorf领衔推荐，资深一线区块链专家联合撰写，区块链和Hyperledger技术扛鼎之作

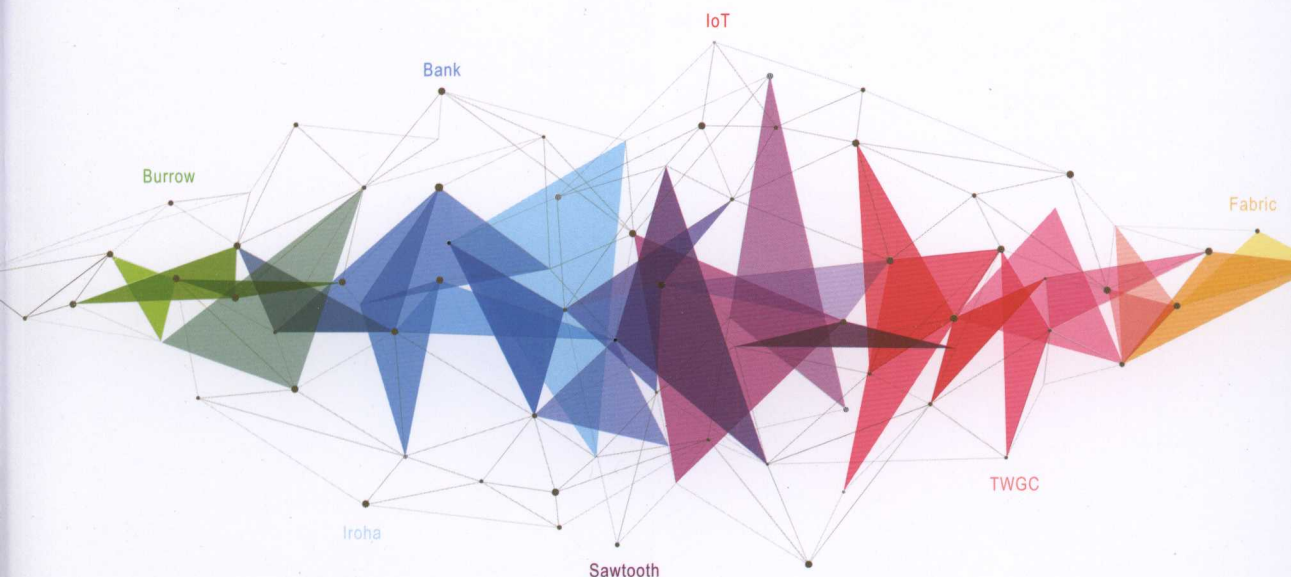
区块链  
技术丛书

深度剖析区块链框架Hyperledger Fabric 1.0的架构、核心技术、部署与应用开发

# 深度探索区块链

## Hyperledger技术与应用

张增骏 董宁 朱轩彤 陈剑雄 ◎ 著



机械工业出版社  
China Machine Press

## 作者简介

---

**张增骏** 智链 ChainNova 技术总监和架构师。

十余年软件开发和项目管理经验，设计并实现了多个区块链项目，带领团队获得“2017 可信区块链峰会”唯一非金融类最佳案例奖。中国信通院可信区块链专家委员会成员，参与讨论并推动可信区块链测试标准的制定，多次受邀到高校、企业分享与推动区块链落地工作。曾任绿盟科技 PDT 经理，带领团队研发的远程安全评估系统（RSAS）连续多年国内排名第一，广泛应用于多个重点领域。目前关注区块链、网络安全、大数据、云计算和人工智能等领域。

**董宁** 智链 ChainNova 科技公司 CEO，北京大学（天津滨海）新一代信息技术研究院金融科技研究中心主任，毕业于北京大学信息科学技术学院智能科学系。曾任 IBM 大中华区 IT 经济学负责人，参与过数家商业银行和金融机构核心系统的设计建设，具有多年金融行业的商业经验。

**朱轩彤** 清华大学硕士，中国社会科学院数量经济与技术经济研究所博士生，专注于技术经济研究。在政府部门及国际组织有丰富的工作经验。

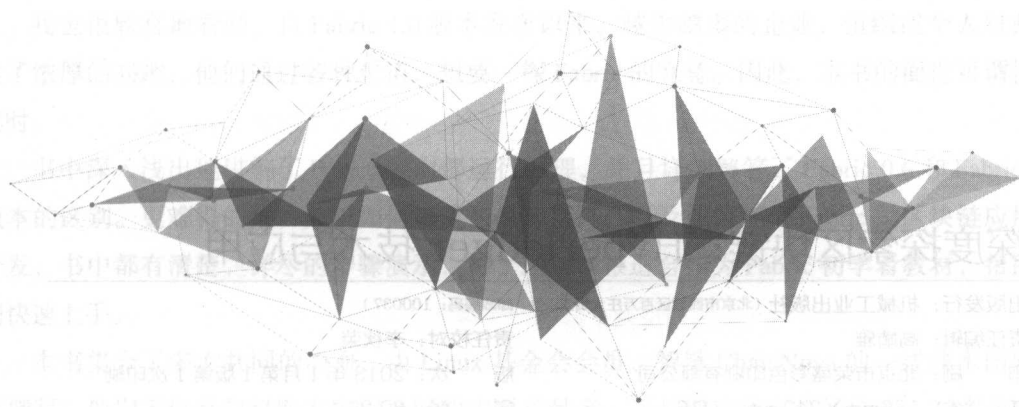
**陈剑雄** 智链 ChainNova 首席科学家，原金山云技术 VP，中科院计算与通信工程学院硕士。多年大规模集群系统研发经验，参与多个区块链核心系统设计和应用。

区块链  
技术丛书

# 深度探索区块链

## Hyperledger技术与应用

张增骏 董宁 朱轩彤 陈剑雄 ◎ 著



机械工业出版社  
China Machine Press

## 图书在版编目(CIP)数据

深度探索区块链: Hyperledger 技术与应用 / 张增骏等著. —北京: 机械工业出版社, 2018.1  
(区块链技术丛书)

ISBN 978-7-111-58932-7

I. 深… II. 张… III. 电子商务 - 支付方付 - 研究 IV. F713.361.3

中国版本图书馆 CIP 数据核字 (2018) 第 003765 号

## 深度探索区块链: Hyperledger 技术与应用

出版发行: 机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码: 100037)

责任编辑: 高婧雅

责任校对: 李秋荣

印 刷: 北京市荣盛彩色印刷有限公司

版 次: 2018 年 1 月第 1 版第 1 次印刷

开 本: 186mm × 240mm 1/16

印 张: 20.25

书 号: ISBN 978-7-111-58932-7

定 价: 79.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88379426 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzit@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光 / 邹晓东

作为 Linux 基金会托管下增长最快的项目，超级账本（Hyperledger）在过去的一年成长十分迅速。这主要归功于 Linux 基金会开放、公平的治理模式，就是让各种规模的组织、开发者社区和技术专家可以达成最高水准的合作，以公开、公平和结构化的方式进行迭代。我们珍视每一个贡献，也鼓励更多的组织和开发者加入其中。

2017 年 7 月，超级账本（Hyperledger）迎来了 Fabric 1.0 版本的正式发布，这是首个可用于生产环境部署的商业级应用，它历经了上百个概念验证。截至发布时间，共有 27 个组织、159 位开发者参与并作出贡献。而在之后的每一天，这个数字一直都在上升。

我也很欣喜地看到，自 Fabric 1.0 版本发布以来，越来越多的企业、组织或个人对此产生了浓厚的兴趣，他们通过各种渠道，想要一探 Fabric 的究竟。因此，本书的面世可谓恰逢其时。

书中深入浅出地讲解了 Fabric 的内部运行原理，并且详细解答了 Fabric 0.6 和 Fabric 1.0 版本的区别。更难得的是，关于如何搭建 Fabric 系统，以及如何基于 Fabric 做区块链应用的开发，书中都有清楚、详尽的步骤演示，易于理解，很适合作为 Fabric 初学者教材，帮助他们快速上手。

本书集合了多方共同的心血，由 Linux 基金会会员、智链 ChainNova 的一线技术团队主笔撰写，他们不仅是超级账本中国社区的主要贡献者，还长期奋战在市场前线，对商业应用环境有相当的了解，相信从书中内容的翔实程度可见一斑。

——Brian Behlendorf，超级账本执行董事

## 序 二 Foreword

区块链技术是当今最具影响力的创新技术之一，得到产业界、学术界和投资领域的广泛关注。

Linux 基金会超级账本（Hyperledger）开源项目创立于 2015 年 12 月，目前已经发展到 160 余个成员单位。超级账本项目的愿景是借助项目成员和开源社区的通力协作，共同制定并建立一个开放、跨产业、跨国界的区块链技术开源标准。它通过创建通用的分布式账本技术，协助组织扩展、建立行业专属应用程序以及平台和硬件系统来支持成员各自的交易业务。

董宁先生这本书深入浅出地描绘了 HyperLedger 技术与应用，为深度探索区块链技术和应用案例提供了很好的参考，是一本不可多得的技术参考书。

本书首先回顾了区块链发展的历史，展望了区块链的商用前景，对超级账本的基础环境、系统架构、交易流程、消息协议、策略管理和访问控制等进行了详细介绍。然后，进一步讨论了 Gossip 协议、分布式账本存储、共识机制、多链和多通道、基于数字证书的成员管理、智能合约等关键技术。最后，介绍了超级账本的应用开发模型、开发案例和应用部署方面的主题。这是我迄今为止所见关于超级账本技术和应用非常有参考意义的一本技术书籍，值得向广大区块链的研究者与开发同行推荐。

董宁先生长期以来致力于区块链技术的研究与推广，也是 Hyperledger 中国社区最有活力的推动者之一。相信本书的出版会对社区的发展和区块链技术的应用起到积极的推动作用。

陈钟

北京大学信息科学技术学院教授

北京大学金融信息化研究中心主任

2017 年 12 月于燕园



## Foreword 序 三

从 2008 年中本聪在论文中提到区块链开始到区块链结合各类产业应用场景落地，区块链以不可思议的速度发展起来，经常会有人在问区块链究竟是什么。可以说，区块链本质上是一种创建信任的技术机制，通过区块链可以跨机构执行可信的交易。

当下和未来，区块链的用武之地将远远超过加密货币，因此为了适应大多数商业应用的需求，设计与开发适合商用的区块链平台迫在眉睫，“超级账本”（Hyperledger）应运而生。作为一个由 IBM 等世界著名大企业领衔的商业化联盟链项目，Hyperledger 是目前代码数量最大、社区参与度最高的区块链开源项目。更重要的是，该项目也标志着区块链从单纯的开源技术发展到了被主流机构和市场认可的阶段。这对于区块链相关产业的发展意义深远。

区块链数学上的可信，不等于工程实现上的可信。为此，中国信通院联合央行数字货币研究所以及 30 多家企业，共同讨论制定了可信区块链标准。2017 年 9 月对包括智链 ChainNova 在内的 9 家企业的区块链进行了第一轮评测，并且于 2017 年 10 月正式在国际标准组织立项。“因为透明，所以可信”，可信区块链标准已经起到了引领和推动我国与全球区块链底层基础设施健康有序发展的目的。在这个过程中，通过与本书作者之一、前 IBM 大中华区 IT 经济学负责人和 IBM 区块链社区发起人、智链 CEO 董宁的接触，能感觉到他对企业级区块链和 Hyperledger 的未来充满信心。对于金融科技和互联网业内人士来说，不懂区块链可能冒着被潮流淹没的风险；对于有志于从事区块链技术的人士来说，不学习 Hyperledger 可能错失与极有可能占据市场领导地位的金融科技结缘的机会。

坦诚地说，本书并非市场上第一本区块链的书。事实上，我了解到在 Hyperledger 0.6 版本盛行之时，本书的作者就曾经完成了本书的初版。但是由于后来 Hyperledger 推出了 1.0 版本，本着对读者极其负责任的态度，他们又全面重写了本书，使读者能够完全跟上 Hyperledger 发展的最新状态。本书的目的也不是蜻蜓点水地介绍一些 Hyperledger 入门知识，

而是通过阅读本书能让读者达到一定的水平，甚至可以加入区块链产业应用中来，为区块链的发展和实践落地添砖加瓦。同时也希望通过作者的努力，能够给有志于在 Hyperledger 平台进行开发，并有所进展的程序开发人员带来帮助。

何宝宏

中国信息通信研究院云计算与大数据研究所所长



## Preface 前言

### 为什么要写这本书

区块链是在全球范围内受到极高关注的技术。简而言之，区块链就是防篡改并且由大家共同维护的账本，其中包含不断增长的数据记录列表。根据现在的发展趋势，区块链将在商业领域得到广泛应用。

超级账本（Hyperledger）是 Linux 基金会旗下的区块链开发平台项目，致力于发展跨行业的商用区块链平台技术。超级账本项目自创立伊始便吸引了众多行业的领头羊，包括金融、银行、互联网、运输、制造等行业。目前，超级账本项目在全球拥有超过 100 个成员，包括 IBM、Cisco、Intel、J.P. Morgan、荷兰银行、SWIFT、R3 等。基于区块链技术、智能合约及其他相关技术，超级账本项目致力于建立新一代的分布式账本交易应用平台，从而在简化与商业流程相关的事务的同时，建立起商业信任、透明、审查等能力。Hyperledger Fabric 子项目是以 IBM 早期捐献出的 Open Blockchain 为主体搭建而成的，是一个带有可插入各种功能模块架构的区块链实施方案，其目标是建立一个更加标准化的开源区块链开发平台，类似 OpenStack 之于云计算。开源地址是：<https://github.com/hyperledger/fabric>。Fabric 主要框架的核心开发语言是 Go 语言，它非常适合联盟链，具有更高的商业应用前景。

从 2015 年开始，由于在 IBM 中国实验室工作，我开始接触区块链技术和 IBM 的 Open Blockchain 项目（即 Hyperledger Fabric 的前身），并开始为中国的金融用户推荐它，帮助用户借助区块链的技术价值来实现科技和业务的创新。到了 2016 年下半年，Hyperledger Fabric 开发平台阶段性地稳定在 0.6 版本，无论是 IBM 还是云图智链（后来被智链 ChainNova 并购），都在很多行业应用场景中开始实践 Fabric 0.6 版本。那时在国内，绝大多数的金融企业都在尝试通过 Hyperledger Fabric 0.6 平台来开发属于自己的区块链应用，我在那个时候有机会参与了不少相关的区块链项目，涉及领域包括数字积分、资产托管和交易、保险、高价值

商品溯源等。也正是从那时起，萌生了编写一本书来解释 Hyperledger 原理，介绍各项开发组件，并通过真实案例还原区块链开发全过程，让更多的人觉得区块链或者 Hyperledger 离自己并不遥远。于是，当时我们几个作者从社区、不同的开发项目，以及各个开发团队中开始收集和整理资料，完成了基于 0.6 版内容的大部分写作工作。

但恰逢此时，Hyperledger 的第一个商用版本 1.0 准备推出，我们也第一时间从 Linux 基金会得到了这个消息。从当时 1.0 版本的计划来看，推出它的目的主要有两个方面：一是 Hyperledger 希望以这个版本为基调，作为企业级区块链平台；二是为了解决 0.6 版本中出现的一些问题，1.0 版本进行了很大的改变和调整，可以说这个变化是翻天覆地的，以至于我们半开玩笑地说从 0.6 版本到 1.0 版本甚至没有可直接升级的路径。当时我还在 IBM 工作，第一时间找到了云图智链的张增骏老师等几位作者和出版社的高婧雅编辑，和他们商量是否需要调整写作方向和内容，因为在此之前实际上张增骏老师已经在上一个版本的写作中付出了很多的时间和精力。几乎是在十分钟之内我们就一致决定要基于 Hyperledger Fabric 1.0 版本重新组织材料，重新编写这本书，因为我们希望自己投入的时间和付出的精力为社区、众多开发者和广大对区块链技术感兴趣的读者带来一本真正实用的书，而不是仅仅为了把我们几个人的名字留在封面上。数月后的今天，我们把初审过的稿件提交给出版社。这个过程中由于 Hyperledger 开源社区和代码版本不断迭代内容也不断调整，经历了更多我们之前没涉及的新行业和应用场景，甚至经历了我个人工作角色的变化，我们几个作者最终坚持完成了本书的写作。当然必须承认，由于能力和经验不足，本书还有很多提升空间，内容本身也难免出现表达不准确的地方。本书希望抛砖引玉，欢迎读者多提宝贵意见，指出本书存在的技术错误，争取在下一版本中能纠正错误，不断完善，进一步提升质量。同时，我们还会一直基于 Hyperledger 这个平台不断地进行产业实践，后面还会以本书为起点策划系列丛书，把我们在工作学习过程中得到的启发和经验分享给读者。

本书作者大多来自智链 ChainNova，均具有多年 IT 工作、实践经验。智链 ChainNova 与超级账本社区一直有非常紧密的合作。2017 年智链 ChainNova 研究院联手 Linux Foundation 和 IBM，共同主办了 HyperLedger Fabric 商用正式版本 1.0 发布后 Hyperledger 开源社区第一次线下会议（北京 Meetup），Hyperledger 全球副总裁、亚太区负责人 Julian Gordan 和多家国内知名金融企业、科技企业高层出席。在 2018 年，智链 ChainNova 正在计划携手 Linux Foundation、北京大学和 IBM 共同主办超级账本黑客马拉松大赛（HyperLedger Hackathon）。同时为了进一步活跃 Hyperledger 社区，我们还即将承办多项开源社区活动，欢迎读者积极参与。相信读者将在学习区块链技术和 Hyperledger 的过程中获益匪浅。

## 本书特色

笔者是 Hyperledger 社区成员，参与超级账本社区的日常工作，了解超级账本发展和技术细节的第一手资料。本书以 Fabric 商用正式版本 1.0 作为底层平台，其中也凝结了我们在 Hyperledger 开发理论和实际操作方面的经验。

本书深入讨论 Hyperledger 的核心技术，帮读者分析原理、关键实现与使用，是为数不多深入探讨和研究区块链的书籍之一。

我们的目标是把本书作为高校、科研院所、职业培训、企业技术学习的教材，向社会普及 Hyperledger，培养更多的 Hyperledger 开发人才。

## 读者对象

- ☐ 区块链从业者
- ☐ 区块链应用开发人员
- ☐ 其他区块链技术爱好者（金融 /Fintec 从业者、产品经理、企业管理者等）
- ☐ 计算机及相关专业师生

## 如何阅读本书

本书共有 12 章内容，大体可以分为三篇内容。

**准备篇**（第 1～2 章），介绍区块链的基本概念，感受区块链的魅力。

**第 1 章** 本章是区块链技术与生态的概览，涉及区块链的基本概念、演进、主流平台，并着重分析企业级区块链平台的应用场景，有助于读者对区块链和 Hyperledger Fabric 1.0（以下简称 Fabric）的设计理念有整体性的了解。

**第 2 章** 本章介绍 Fabric 的安装、部署与调试。抛开复杂的底层技术细节，简化复杂的部署过程，快速体验 Fabric 的强大功能，以便有直观的感受。

**核心篇**（第 3～9 章），从系统架构开始讲解内部实现机制。

**第 3 章** 本章基于 Fabric 1.0 讲解区块链的架构，这是后续章节的基础，高屋建瓴地看待各个部件之间的关系和运行逻辑。本章涉及系统逻辑架构、网络节点架构、典型的交易流程、消息协议结构、策略管理和访问控制等内容，后面章节会从技术角度逐一“拆解”。

**第 4 章** 本章介绍基于 Gossip 的 P2P 数据分发机制，包括节点启动与成员管理、主节点选举与基于反熵的状态同步、身份认证与管理、多路分用与分区处理过程、消息的多种验证

策略等。

**第5章** 本章介绍最为基础的分布式账本技术，它涵盖账本数据、索引数据、状态数据、历史数据等的实现技术。

**第6章** 本章介绍如何在排序服务上实现多通道的数据隔离，包括创建通道、节点加入通道等。排序服务采用插件化设计，可以根据业务场景的需求采用不同的共识算法。本章后面的内容详细介绍了排序服务的接口，以及实现了排序服务接口的 Solo 和 Kafka 模式。

**第7章** 本章介绍 Fabric 1.0 支持的多链及其内部的实现，多个链同时运行是一个系统工程，本章从数据存储、链码、命令行工具和 SDK 实现等多个方面分析如何支持多链。

**第8章** 本章介绍成员管理机制。它分为两个部分，第一部分详细介绍了 MSP 机制，包括 MSP 成员的验证、目录结构和配置最佳实践等；第二部分介绍可选的 Fabric CA，包括服务端的安装部署和客户端的使用，还介绍了服务端提供的 RESTful 接口。

**第9章** 本章介绍 Fabric 1.0 上智能合约的实现。包括的内容有链码的生命周期管理、内置的系统链码、链码的相互调用、背书节点和链码的有限状态机等。

**应用篇（第10～12章）**，从安装部署和应用开发的角度，通过一个票据背书的案例讲解如何基于 Hyperledger Fabric 1.0 开发区块链应用。

**第10章** 本章介绍 Fabric 1.0 的应用开发模型。从应用开发的角度看，开发者需要关注两部分：一部分是基于不同语言的 SDK 开发和区块链网络交互的应用程序；另一部分是实现超级账本的智能合约。本章详细介绍 HFC SDK 各个模块及其主要功能，链码的主要接口及其功能。

**第11章** 本章介绍多种 Fabric 1.0 的部署方式，包括分别基于 Vagrant、Virtualbox、Docker 的运行环境，以及 BYFN 脚本的使用。详细说明如何手动构建 Fabric 1.0 网络等。

**第12章** 本章通过一个票据背书示例，讲解如何实际开发一个基于 Fabric 1.0 的区块链应用。通过本章的实践，读者能够掌握区块链应用开发的方方面面，然后就可以动手开发具体的项目了。

## 读者反馈与勘误

欢迎读者朋友反馈，请让我们知道你对本书的看法——你喜欢哪些地方，不喜欢哪些地方。读者反馈对于我们很重要，因为这将帮助我们继续写作使你获益的书籍。反馈意见请发送 E-mail 至 [jessie@chainnova.com](mailto:jessie@chainnova.com)，并在邮件主题中指明书名，我们将尽力解决问题。如果你有专长领域，并对写书或为书做出其他贡献感兴趣，请访问 [www.chainnova.com](http://www.chainnova.com) 参见作者指南。

## 特别致谢

首先感谢本书的其他作者——张增骏老师、朱轩彤老师和陈剑雄老师。他们在工作之余，挤出宝贵时间为本书贡献了他们对区块链技术和 Hyperledger 的理解和洞察。特别感谢张增骏老师在工作本身比较繁忙的前提下，为本书花费了很多精力，他不仅在内容上积极供稿，还在审定、修改和校正方面下了很多工夫。朱轩彤老师博闻强识，本身具有很强的行业背景，对科技产业的发展又格外关注，这些在本书第 1 章中得到了充分体现。智链首席科学家陈剑雄也对本书的内容给出了很多宝贵的意见和建议，同时对本书合作的达成给予了支持。

万分感谢超级账本执行董事 Brian Behlendorf 先生，北京大学陈钟教授和中国信息通信研究院云计算与大数据所何宝宏所长在百忙之中拨冗为本书做序，让我感觉特别荣幸。他们在各自领域都是最顶尖的专家，同时对区块链技术都有深刻且独到的见解。还有苏州同济金融科技研究院马小峰院长、中国电子学会区块链专委会孙贻滋秘书长和超级账本中国技术工作组杨保华主席为本书写来热情洋溢的推荐，令人备感温暖。

在成书的过程中，和我一起工作和合作的很多专家对本书都给予了不同程度的支持和帮助，像 Linux 基金会超级账本亚太区副总裁 Julian Gordon 和中国地区顾问龙文选先生，北京大学（天津滨海）新一代信息技术研究院马修军副院长，中国信息通信研究院云计算与大数据所魏凯主任和卿苏德博士，IBM 的各位领导和专家，以及其他各个单位的领导和大咖，在此抱歉不能一一尽述。

非常感谢机械工业出版社华章公司的编辑高婧雅，她的敬业精神和编辑效率令我由衷敬佩，她的反馈、建议、鼓励和帮助引导我们克服诸多困难完成全部书稿。同时，本书的推广得到了 CSDN 及其副总裁孟岩先生、InfoQ 及其总编辑郭蕾先生这些好朋友的大力支持。

最后，因为工作和写作，牺牲了很多本该陪伴家人的时间。我要特别感谢我的家人长期以来对我的默默支持和理解。

谨以本书献给我最亲爱的家人，多年以来帮助、支持我的师友，以及众多热爱区块链技术的朋友！

董宁

2017 年 12 月



## 目 录 Contents

序一	1.3.2 以太坊·····	11
序二	1.3.3 瑞波·····	13
序三	1.3.4 区块链商用平台：超级账本·····	13
前言	1.3.5 区块链技术平台比较·····	15
	1.4 区块链的商用之道·····	15
	1.4.1 区块链的 2.0 时代：商用 区块链·····	15
	1.4.2 超级账本：商用区块链的 “第五元素”·····	17
	1.4.3 区块链的商业应用场景·····	17
	1.5 本章小结·····	18
第一篇 准备篇	第2章 超级账本初体验·····	19
第1章 区块链概述·····	2.1 基础环境安装·····	19
1.1 区块链的前世今生·····	2.1.1 Docker 的安装和使用·····	19
1.1.1 区块链的历史起源——比特币·····	2.1.2 Docker Compose 的安装和 使用·····	21
1.1.2 欢迎来到区块链的世界·····	2.1.3 下载超级账本源代码·····	24
1.1.3 区块链演进趋势·····	2.2 超级账本部署调用·····	24
1.2 区块链概念·····	2.2.1 下载 Docker 镜像文件·····	24
1.2.1 区块链本质·····	2.2.2 部署超级账本网络·····	25
1.2.2 区块链工作原理·····	2.2.3 链码调用和查询·····	26
1.2.3 区块链技术特点·····		
1.2.4 区块链层次模型·····		
1.2.5 区块链共识算法·····		
1.2.6 区块链并不一定去中心化·····		
1.3 区块链技术平台·····		
1.3.1 比特币·····		

2.2.4 常见错误	27
2.3 节点的配置参数传递规则	29
2.4 本章小结	31

## 第二篇 核心篇

<b>第3章 超级账本的系统架构</b>	34
3.1 系统逻辑架构	35
3.2 网络节点架构	37
3.3 典型交易流程	39
3.3.1 创建交易提案并发送给背书节点	39
3.3.2 背书节点模拟交易并生成背书签名	41
3.3.3 收集交易的背书	42
3.3.4 构造交易请求并发送给排序服务节点	43
3.3.5 排序服务节点以对交易进行排序并生成区块	45
3.3.6 排序服务节点以广播给组织的主节点	45
3.3.7 记账节点验证区块内容并写入区块	45
3.3.8 在组织内部同步最新的区块	49
3.4 消息协议结构	49
3.4.1 信封消息结构	49
3.4.2 配置管理结构	51
3.4.3 背书流程结构	52
3.5 策略管理和访问控制	56
3.5.1 策略定义及其类型	56
3.5.2 交易背书策略	57

3.5.3 链码实例化策略	60
3.5.4 通道管理策略	61
3.6 本章小结	63
<b>第4章 基于Gossip的P2P数据分发</b>	64
4.1 概述	64
4.2 超级账本中的 Gossip 协议	65
4.3 成员认证及身份管理	67
4.4 节点启动及成员管理	67
4.5 主节点选举过程	68
4.6 基于反熵的状态同步	69
4.7 数据传播过程	70
4.8 多通道的支持	70
4.9 消息的验证策略	71
4.10 消息的多路分用及分区	73
4.11 和 Gossip 相关的配置参数	76
4.12 本章小结	77
<b>第5章 分布式账本存储</b>	78
5.1 概述	78
5.2 读写集	79
5.2.1 交易模拟和读写集	79
5.2.2 交易验证和世界状态更新	80
5.2.3 模拟和验证示例	80
5.3 账本编号	81
5.4 账本数据	81
5.4.1 账本数据存储	82
5.4.2 账本数据读取	83
5.4.3 交易模拟执行	84
5.5 区块索引	84
5.5.1 文件位置指针	85

5.5.2	索引的同步过程	86
5.6	状态数据	87
5.6.1	LevelDB	88
5.6.2	CouchDB	89
5.6.3	基于状态数据的区块验证	91
5.7	历史数据	92
5.8	数据恢复	92
5.9	本章小结	93
<b>第6章</b>	<b>集成共识机制的排序服务</b>	<b>94</b>
6.1	概述	94
6.1.1	共识算法的类型	95
6.1.2	Hyperledger Fabric 1.0 的 共识机制	96
6.2	实现数据隔离的多通道	97
6.2.1	排序服务的初始化	99
6.2.2	通道的创建	101
6.2.3	通道的更新	105
6.2.4	通道的加入	107
6.2.5	通道的查询	107
6.3	可插拔的排序服务	108
6.3.1	排序服务接口	108
6.3.2	基于单进程的排序服务	110
6.3.3	基于 Kafka 的排序服务	110
6.3.4	链消息过滤器	122
6.4	本章小结	124
<b>第7章</b>	<b>实现数据隔离的多链及 多通道</b>	<b>125</b>
7.1	数据存储对多链的支持	126
7.1.1	账本数据	126

7.1.2	索引数据	126
7.1.3	状态数据	127
7.1.4	历史数据	127
7.2	链码对多链的支持	128
7.2.1	链码的生命周期管理	128
7.2.2	链码和背书节点的通信	129
7.2.3	链码的部署和调用	130
7.3	多通道对多链的支持	131
7.4	命令行和 SDK 对多链的支持	132
7.5	关于系统链	132
7.6	本章小结	132
<b>第8章</b>	<b>基于数字证书的成员管理 服务</b>	<b>133</b>
8.1	实现成员管理的 MSP	133
8.1.1	MSP 成员的验证	133
8.1.2	MSP 的目录结构	134
8.1.3	MSP 的配置最佳实践	140
8.2	颁发数字证书的 Fabric CA	142
8.2.1	概述	142
8.2.2	Fabric CA 服务端的安装 部署	143
8.2.3	Fabric CA 服务端的操作 使用	148
8.3	本章小结	158
<b>第9章</b>	<b>支持多种语言的智能合约</b>	<b>159</b>
9.1	概述	160
9.2	链码的生命周期管理	160
9.2.1	链码的生命周期	160
9.2.2	应用程序和链码的交互 流程	164



9.2.3	背书节点接收应用程序的请求处理	165
9.2.4	采用上下文实现交易的模拟执行	166
9.2.5	链码消息的数据分发	166
9.2.6	链码运行环境的管理	168
9.3	内置的系统链码	172
9.3.1	生命周期管理系统链码	173
9.3.2	配置管理系统链码	180
9.3.3	查询管理系统链码	182
9.3.4	交易背书系统链码	182
9.3.5	交易验证系统链码	184
9.4	链码的相互调用	184
9.5	背书节点和链码的有限状态机	185
9.5.1	背书节点和链码之间的事件	188
9.5.2	背书节点的有限状态机	189
9.5.3	链码的有限状态机	190
9.6	本章小结	192

### 第三篇 应用篇

#### 第10章 超级账本的应用开发模型 194

10.1	应用开发模型	194
10.2	应用程序开发的 SDK	194
10.2.1	概述	195
10.2.2	SDK 规范	195
10.2.3	应用场景介绍	204
10.3	链码的开发和调试	210
10.3.1	链码需要实现的接口	210

10.3.2	链码的 SDK 提供给链码的接口	212
10.3.3	链码开发的注意事项	214
10.3.4	链码的调试	215
10.4	本章小结	216

#### 第11章 从零开始部署超级账本网络 217

11.1	准备超级账本运行环境	217
11.1.1	超级账本运行环境	217
11.1.2	编译超级账本镜像文件	224
11.2	快速构建超级账本网络	227
11.2.1	下载 BYFN 的代码	227
11.2.2	BYFN 脚本介绍	227
11.2.3	生成网络初始化配置	228
11.2.4	启动超级账本网络	230
11.2.5	关闭超级账本网络	235
11.3	逐步建立超级账本网络	236
11.3.1	生成 MSP 证书	236
11.3.2	生成排序服务创世区块	236
11.3.3	生成通道配置创世区块	236
11.3.4	定义组织锚节点	237
11.3.5	启动超级账本网络	237
11.3.6	创建并加入通道	238
11.3.7	安装和实例化链码	240
11.3.8	执行链码查询	243
11.3.9	执行链码调用	244
11.4	本章小结	245

#### 第12章 超级账本的应用开发实例 246

12.1	票据背书场景介绍	246
------	----------	-----

12.1.1 票据关系人 ..... 247

12.1.2 票据行为分类 ..... 247

12.1.3 基于区块链技术的数字  
票据 ..... 249

12.2 票据背书需求分析 ..... 250

12.3 票据背书架构设计 ..... 251

12.3.1 票据背书的分层架构 ..... 252

12.3.2 票据背书的数据模型 ..... 253

12.4 票据背书实现 ..... 254

12.4.1 应用程序实现 ..... 254

12.4.2 链码功能实现 ..... 275

12.5 票据背书快速部署 ..... 287

12.6 票据背书展示 ..... 288

12.6.1 系统登录 ..... 288

12.6.2 发布票据 ..... 288

12.6.3 我的票据 ..... 289

12.6.4 发起票据背书 ..... 289

12.6.5 待签收票据列表 ..... 290

12.6.6 签收票据背书 ..... 290

12.6.7 拒收票据背书 ..... 291

12.7 本章小结 ..... 292

附录A 术语表 ..... 293

附录B 超级账本的实用工具 ..... 297

参考文献 ..... 308

## 第一篇 Part 1

# 准备篇

### ■ 第1章 区块链概述

### ■ 第2章 超级账本初体验

# 区块链概述

## 1.1 区块链的前世今生

区块链的发展历史比较短暂，最初仅仅作作为支持数字货币比特币交易的技术。目前，区块链技术已经脱离比特币，在金融、贸易、征信、物联网、共享经济等诸多领域得到初步应用。由于区块链技术可以防止数据篡改，所以不仅可以用安全而透明的方式追踪比特币的活动，还能在区块链网络中追踪其他类别的数据，因此可以帮助私人公司或政府部门建立更值得信赖的网络。用户可在这个网络中分享信息和价值，未来它还将得到更广泛的应用。

### 1.1.1 区块链的历史起源——比特币

比特币起源于 2008 年全球金融危机期间中本聪（Satoshi Nakamoto）撰写的论文《Bitcoin: A peer-to-peer electronic cash system》（《比特币：一种点对点的电子现金系统》）<sup>[1]</sup>。在这篇著名的论文中，重点讨论了比特币系统，区块链被描述为用于记录比特币交易的账目历史。事实上，中本聪关于比特币的白皮书虽然整体思想是开创性的，但其中使用的技术工具，如 P2P（Peer-to-Peer）、分布式存储、非对称性加密等早已存在，他提出的是一个集成性的、系统性的、可供实践的解决方案。

实际上在这篇论文中并没有明确提出区块链的定义和概念，甚至还没有“区块链”（Blockchain）这个词，只有“区块”（Block）和“链”（Chain）。但在论文中涉及了几个对区块链技术影响深远的观点：

- 点对点、去中心化的可靠交易；
- 反欺诈；

- 基于密码学原理的电子交易凭证管理；
- 分布式的时间戳服务器；
- 足够的安全能力支持系统。

在2009年他公开了最初的实现代码，第一个比特币于2009年1月3日18:15:05生成，但真正流行起来是2010年后的事情。设计和实现一种数字货币绝非易事，但是由于比特币设计精妙，诞生还不到十年，就在世界各地迅猛发展，从小众成功走向主流。

在日本和韩国比特币的交易特别火爆，比特币价格在震荡中不断刷新纪录。2009年10月最早有纪录的比特币价格仅为0.00076美元。不到8年时间，2017年上半年比特币这个全球最热门的加密货币的价格创历史新高，首次超过2000美元。收益远远超过黄金、地产，甚至有人预测其价格还将大幅攀升。此外，2017年5月勒索病毒WannaCry席卷全球，全球150个国家及地区的数十万台计算机遭到攻击，事件并没有给比特币市场造成直接的损失，反倒让比特币也跟着火了一把。

据报道，多国政府开始给予比特币合法身份。2017年4月，日本正式生效新规，比特币成为一种合法的支付方式；2017年7月，澳大利亚政府承认比特币为货币，并废除比特币的商品与服务税。

### 1.1.2 欢迎来到区块链的世界

在比特币系统成功运行多年后，2014年前后，部分金融机构开始意识到，作为比特币运行的底层支撑技术——区块链，实际上是一种极其巧妙的分布式共享账本技术，对金融乃至各行各业带来的潜在影响甚至可能不亚于复式记账法的发明。对区块链的初步认识来自2014年10月大英图书馆的一次技术讨论会。在这次会议中，人们对比特币的现状和未来以及区块链在金融等领域的应用前景进行了深入的探讨。

自此，区块链技术开始在全球崭露头角。可以把2015年称为世界区块链元年，因为在这一年经济领域关注到了它。划时代的标志是《华尔街日报》刊文称区块链是最近500年以来在金融领域最重要的突破，而《经济学人》杂志在封面《信任的机器》一文中介绍区块链为创造信任的机器。文章指出，区块链并非仅仅是一项加密技术或者数字货币，在信息不对称、不确定的环境下，它还可以建立满足经济活动赖以发生、发展的“信任”生态体系。作为比特币底层技术的“链”，其价值远大于比特币本身。区块链可以让人们在没有中央权威机构监督的情况下，对彼此协作建立起信心。区块链是一种共享账本技术，实现了在分布式商业网络里多方参与的双边交易中的去中介化。简单来说，它是一台创造信任的机器。

进入2016年，业界开始大规模认识到区块链技术的重要价值，并通过智能合约技术将其用于数字货币以外的分布式应用领域。世界经济论坛甚至预测，到2025年，世界GDP的10%都将存储在区块链上或者应用区块链技术。

政府部门也行动起来关注区块链。2016年1月，英国首席科学家建议英国政府把区块

链技术列为英国国家战略，随后包括中国人民银行在内的多国政府和央行对区块链进行了研究。2017年，欧洲议会发布了一份新的报告——《区块链如何改变我们的生活》，总结了区块链技术的能力和以及对社会价值可能带来的影响，其中特别指出智能合约的重要作用。

虽然直到今天，我们还不知道中本聪究竟是谁，但毫无疑问他开启了一个新的时代。许多商业组织和行业机构投入了大量资源来研究区块链。全球领先的信息技术研究和顾问公司 Gartner 公布了 2016 年新兴科技技术成熟度曲线（Hype Cycle for Emerging Technologies, 2016）<sup>[2]</sup>，如图 1-1 所示。2016 年区块链正处于期望膨胀期，距离成熟期需要 5 ~ 10 年。在未来，全球区块链技术仍然会保持比较高的发展趋势。

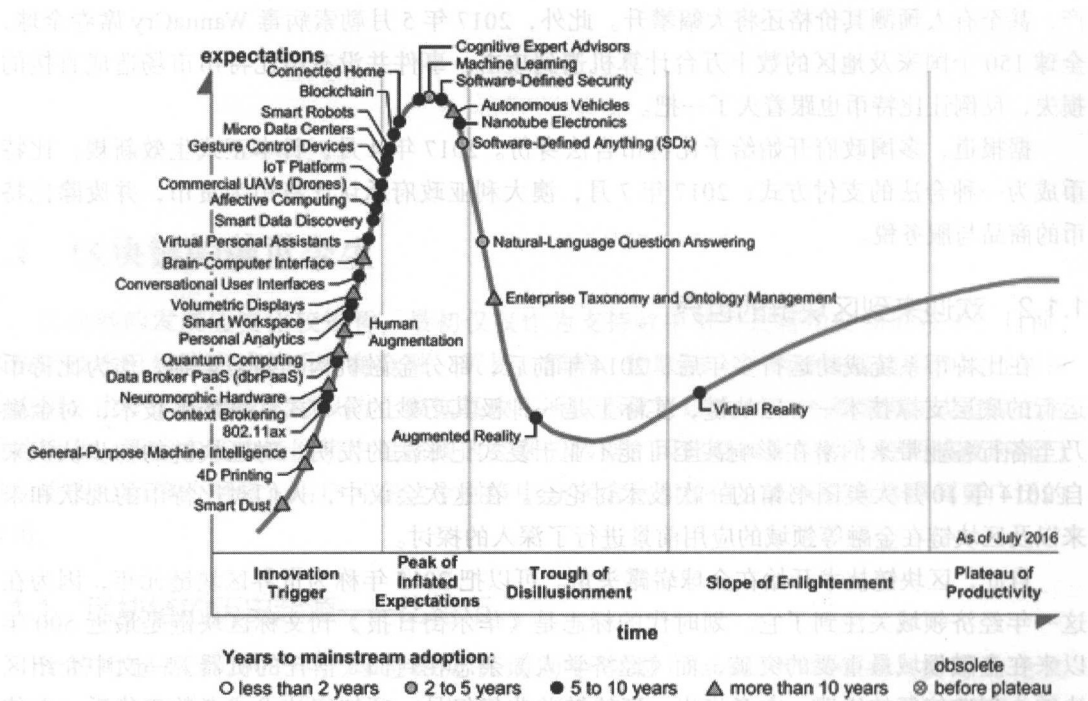


图 1-1 2016 年新兴科技技术成熟度曲线

### 1.1.3 区块链演进趋势

本质上，因为区块链的链与链之间具有隐私、安全、共识、自治、价值共享的特性，所以在技术层面上解决了互联网上的价值传递问题。同时，区块链又具有底层开源和改变业务规则、创新业务多方共识等逻辑，因此区块链是未来整个 IT 架构和互联网转型的重要支撑。

Melanie Swan 所著的《区块链：新经济蓝图及导读》<sup>[3]</sup>一书在业界引起了巨大的反响。她在书中提出了对区块链版本划分的方法，即按照区块链已经完成的以及将要完成的功能

划分成区块链 1.0、2.0 和 3.0 三个阶段。这种版本划分方式基本上反映了区块链技术成熟发展的大脉络，目前也得到了业界广泛的认可。

1) 我们可以把比特币理解为区块链技术的一个应用场景，也就是区块链 1.0 阶段。但是如果仅有比特币，区块链也只是一种数字货币，并不能达到今天的火爆程度，可以说比特币是当今区块链的“杀手级”应用，但是区块链可以做的事情远远超过比特币，很有可能产生其他“杀手级”应用。

2) 区块链 2.0 的重要标志就是被金融领域所接受并得到广泛应用形成的金融互联网，让价值交换变得便捷、直接，节省时间、节省成本。目前区块链 2.0 在实际场景中的应用，有两个重要因素：资产数字化（上链过程）与智能合约。

区块链 2.0 更关注智能合约（Smart Contract）所体现的业务价值。在区块链的背景下，智能合约当作是一种运行在区块链之上的通用计算模式，这样智能合约的内涵就不一定必须要和传统的合同概念相关联，反而可以是任何的计算机程序。智能合约实际上是通过高级编程语言把现实世界的业务逻辑在区块链上加以实现。智能合约通过在区块链上增加应用功能拓展了其适用范围和生存空间，如此就可以通过区块链来描述众多实现当中的业务场景。

当前，技术和产业处于区块链 2.0 阶段。在摩根士丹利公司的一份报告中提到了他们对区块链技术在金融行业被采用的路线图展望，对区块链 2.0，其展望大致持续到 2025 年：

- 2014 年—2016 年是评估阶段。银行和其他金融基础设施中介机构对许可制的共享账本技术的效率、机会等进行评估；
- 2016 年—2018 年对区块链进行概念原型验证测试。主要的测试目标是验证技术的可行性，将区块链技术和传统方式在性能、成本、速度、规模等方面进行对比；
- 2017 年—2020 年预计基于区块链的共享架构开始出现；
- 2021 年—2025 年在区块链技术证明有效的基础上，会有更多的金融资产转向区块链技术。

3) 区块链 3.0 要把区块链的应用范围拓展到各行各业，支持广义的资产交互和登记，进入万物互联，设备民主的“区块链+”时代。

互联网使得全球之间的互动越来越紧密，伴随而来的是巨大的信任鸿沟，未来将进入到需要真正的强信任背书的大数据时代。通过使用区块链技术，任何人都没有能力也没有必要去质疑数据的质量和真实性。区块链技术具有全新的理念和逻辑结构，并且它每天还在发展变化过程中，因此，随着区块链能够为信任提供价值的场景改变，它自身也将进入不同的阶段。或许未来的某一天，区块链可能还将迈进更新的阶段。

## 1.2 区块链概念

区块链（Blockchain）技术自身仍然在飞速发展中，目前还缺乏统一的规范和标准。Wikipedia 给出的定义为：



A blockchain, originally block chain, is a distributed database that maintains a continuously-growing list of data records hardened against tampering and revision. It consists of data structure blocks—which hold exclusively data in initial blockchain implementations, and both data and programs in some of the more recent implementations—with each block holding batches of individual transactions and the results of any blockchain executables. Each block contains a timestamp and information linking it to a previous block.

简而言之，区块链技术让参与的系统中任意多个节点，通过密码学算法把一段时间系统内的全部信息交流数据计算和记录到一个数据块（Block）中，并且生成该数据块的指纹用于链接（Chain）下个数据块和校验，系统中所有的参与节点共同认定记录是否为真。

### 1.2.1 区块链本质

区块链，实质是由多方参与共同维护的一个持续增长的分布式数据库，也称为分布式共享账本（Distributed Shared Ledger），其核心在于通过分布式网络、时序不可篡改的密码学账本及分布式共识机制建立彼此之间的信任关系，利用由自动化脚本组成的智能合约来编程和操作数据，最终实现由信息互联向价值互联的进化，如图 1-2 所示。

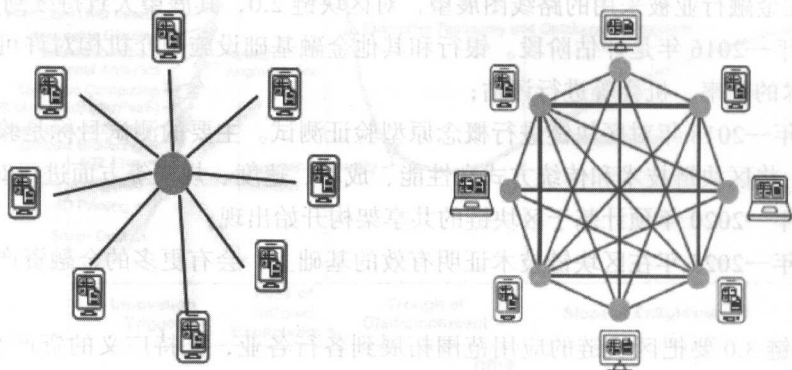


图 1-2 从传统集中记账方式（左）到分布式总账（右）

区块链是一种与传统集中记账方式不同的记录技术。参与到区块链系统上的节点，可能不属于同一组织、彼此无须信任；区块链数据由所有节点共同维护，每个参与维护的节点都能获得一份完整记录的拷贝。与传统的记账技术相比，其特点包括：维护一条不断增长的链，只可能添加记录，而发生过的记录不可篡改；无须集中控制而能达成共识，实现上尽量采用分布式；通过密码学的机制来确保交易无法抵赖和破坏，并尽量保护用户信息和记录的隐私性。

### 1.2.2 区块链工作原理

所谓区块链，正是由多个区块组成的链状数据结构及存储方式。每个区块分为区块头



和区块体，区块头主要用来实现区块链接的前一区块哈希值（Hash Value），而区块体主要包括交易账本，如图 1-3 所示。

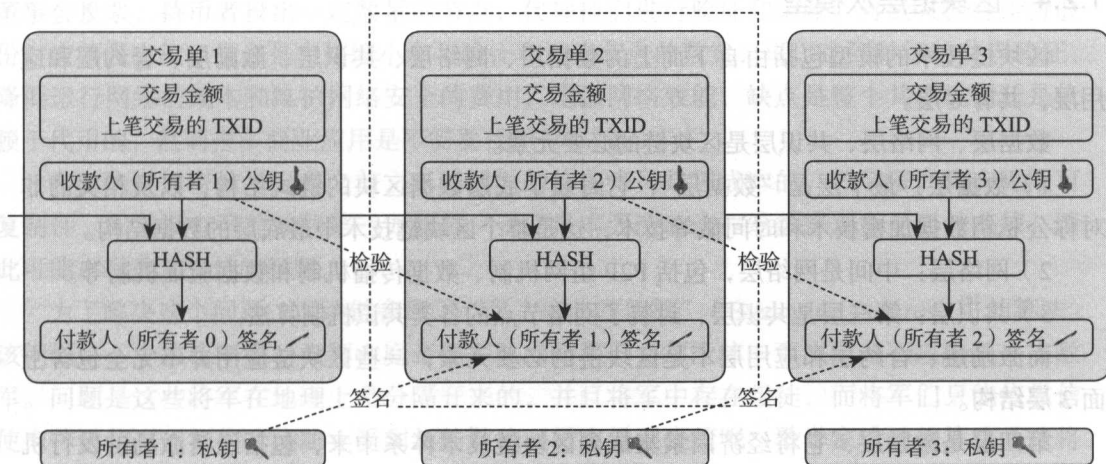


图 1-3 区块链的账本

以交易场景为例，区块链的工作原理如下：

- 1) 客户端将发起一笔交易，经数字签名后广播给网络上的其他节点并等待确认；
- 2) 网络中的节点对收到的数据记录信息进行校验，通过校验后，数据记录被记录到一个区块中；
- 3) 全网所有接收节点对区块执行共识算法，区块通过共识算法过程后正式纳入区块链中存储，全网节点均表示接受该区块。表示接受的方法，是将该区块的随机哈希值视为最新的区块哈希值，新区块将提供永久和透明的交易记录并以该区块链为基础进行延长，实现资金转移。

### 1.2.3 区块链技术特点

具体来说，区块链技术作为创造信任的机器，主要有以下特点。

- ❑ **分布式结构。**区块链构建在分布式网络基础之上，账本并不是集中存放在某个服务器或数据中心，也不是由第三方权威机构来负责记录和管理，而是分散在网络中的每一个节点上，每一个节点都有一个该账本的副本，所有副本同步更新。
- ❑ **信任机制。**区块链技术通过数学原理和程序算法，使系统运作规则公开透明，实现交易双方在不需要借助第三方权威机构信用背书下通过达成共识，建立信任关系。
- ❑ **公开透明。**区块链对其上的节点可以做到开放、透明。任何人都可以加入区块链，也能查询区块链上的区块记录；同时所有用户看到的是同一个账本，能看到这个账本所发生和记录的每一笔交易。
- ❑ **时序不可篡改。**区块链采用带有时间戳的链式区块结构存储数据，具有极强的可追

溯性和可验证性；同时由密码学算法和共识机制保证了区块链的不可篡改性。

#### 1.2.4 区块链层次模型

区块链技术的模型包括由自下而上的数据层、网络层、共识层、激励层、合约层和应用层，共有6层。

数据层、网络层、共识层是区块链的必要元素。

1) 数据层：最下层是“数据层”，它封装了底层数据区块的链式结构，以及相关的非对称公私钥数据加密技术和时间戳等技术，这是整个区块链技术中最底层的数据结构。

2) 网络层：中间是网络层，包括P2P组网机制、数据传播机制和数据验证机制等。

3) 共识层：第三层是共识层，封装了网络节点的各类共识机制算法。

而激励层、合约层和应用层不是区块链的必要元素，一些区块链应用并不完全包含上面3层结构。

第四层是激励层，它将经济因素集成到区块链技术体系中来，包括经济激励的发行机制和分配机制等，主要出现在公有链当中。

第五层是合约层，它封装各类脚本、算法和智能合约。

第六层是应用层，它封装了区块链的各种应用场景和案例，未来的可编程金融和可编程社会也将搭建在应用层中。

#### 1.2.5 区块链共识算法

区块链并不是某种特定技术，而是一种类似于NoSQL（非关系型数据库）这样的技术解决方案的统称。共识机制算法是区块链的核心技术，因为这决定了到底由谁来记账，而记账方式将会影响整个系统的安全性和可靠性。目前已经出现了十余种共识机制算法，其中较为知名的有PoW(Proof of Work, 工作量证明), PoS(Proof of Stake, 权益证明), DPoS(Delegate Proof of Stake, 股份授权证明) 机制以及拜占庭将军容错共识等。当然，没有一种共识机制是完美无缺的，同时这也意味着没有一种共识机制是适合所有应用场景的。

1) PoW（工作量证明）：就是挖矿，主要应用包括比特币和以太坊前三个阶段等。依赖机器进行数学运算来获取记账权，即通过与或运算，计算出一个满足规则的随机数，获得本次记账权，发出本轮需要记录的数据，全网其他节点验证后一起存储。优点是完全去中心化，节点自由进出；缺点是资源消耗相比其他共识机制高、可监管性弱，同时每次达成共识需要全网共同参与运算，性能效率比较低，达成共识的周期较长，因此不适合商业应用。

2) PoS（权益证明）：由Quantum Mechanic 2011年在bitcointalk首先提出，它是PoW的一种升级共识机制，在Peercoin、NXT和以太坊第四个阶段等应用。根据每个节点所占代币的比例和时间，等比例地降低挖矿难度，从而加快寻找随机数的速度，因此节点记账权的获得难度与节点持有的权益成反比，但它依然是基于哈希运算竞争获取记账权的方式。其优点是相对于PoW在一定程度减少了数学运算带来的资源消耗，性能也得到了相应的提

升；缺点是还需要挖矿，本质上没有解决商业应用的痛点，可监管性也比较弱。

3) DPoS (股份授权证明)：与 PoW 和 PoS 不同，DPoS 不需要再挖矿了，而是类似于董事会投票，持币者投出一定数量的节点，代理他们进行验证和记账，持股人拥有所持股份对应的表决权。优点是大幅缩小参与验证和记账节点的数量，可以达到秒级的共识验证，降低运行网络的成本和维护网络安全费用，增强网络效能；缺点是整个共识机制还是依赖于代币的，然而很多商业应用是不需要代币存在的。

在区块链加密技术出现之前，互联网上的信息拷贝是零成本的，数字资产具有无限可复制性，如果没有可信赖的第三方监督，我们根本无法确认一笔数字现金是否被花掉，因此可能出现重复支付的问题。

为了解决这个问题，区块链参照了“拜占庭将军问题” (Byzantine failures)<sup>[5]</sup> 的算法。该问题是一个协议问题，指拜占庭帝国军队的将军们必须全体一致决定是否攻击某一支敌军。问题是这些将军在地理上是分隔开来的，并且将军中存在叛徒，而将军们只能依靠信使来传递信息。如何才能防止受到叛徒欺骗而做出错误决策呢？数学家设计的算法是让将军在接到上一位将军标有进攻时间的信件之后，写上同意或反对并盖上自己的图章，然后把信转发给其他所有的将军，在这样的信息周转之后，最后会出现一个盖有超过半数将军图章的信息链，以保证将军们在互不信任的情况下达成共识。

莱斯利·兰伯特把拜占庭将军问题引入到点对点通信中。拜占庭假设是对现实世界的模型化，由于硬件错误、网络拥塞或断开以及遭到恶意攻击，计算机和网络可能出现不可预料的行为。拜占庭容错协议必须处理这些失效，并且这些协议还要满足所要解决问题要求的规范。

区块链的技术原理参考了拜占庭将军问题的算法，通过盖戳的形式来进行公证。网络上的每一个参与者的计算机里都会有一份总账的备份，也都能在这本总账里记上一笔，并且所有的备份都是在实时地、持续地进行更新、对账，以及同步着拷贝，即全网记账。每个节点都可以竞争盖戳，互相认证。这使得一个不可信网络变成了一个可信的网络，使得所有参与者可以在某些事情上达成一致。

### 1.2.6 区块链并不一定去中心化

理想化的区块链系统，是由许许多多节点组成的点与点的网络结构，似乎既不需要中心化的硬件设备，也不需要任何管理它的机构。在很多文献中都提出区块链是去中心化的 (Decentralized)，即整个网络没有中心化的硬件或者管理机构，任意节点之间的权利和义务都是均等的，且任一节点的损坏或者失去都不会影响整个系统的运作。

需要指出的是，区块链并不一定是去中心化的。实际上，软件系统的网络架构一般有 3 种模式：单中心、多中心、分布式，Decentralized 只表明不是单中心模式的，它可能是多中心或弱中心，也可能是分布式的。把 Decentralized 翻译成“去中心化的”可能与最早由中国大陆“币圈”所做出的翻译偏差有关，实际上在中国台湾地区大多译为“分散式的”。

而不是“去中心化的”。因此，把 Decentralized 翻译成“分布式”或者“多中心化”，可能更切合今天技术和金融场景应用的实际。

在人类历史上，信息传播的延迟和谬误会导致信息不对称，这形成了各种中心化的强权组织和阶级分化。虽然信息社会特别是社交媒体的出现已经减弱了这种情况，但是目前仍无法达到绝对的信息对称。我们期待在区块链搭建的机器社会中进行深刻且迅速的社会关系变革，形成绝对信息对称，但是至少目前在机器社会还难以实现，即不能完全去中心化。

2016 年 The DAO 受攻击事件也表明，完全去中心化至少在现阶段是不可行的。The DAO 是一个基于以太坊公有链的众筹项目，成为史上最大的众筹项目。然而由于其智能合约的漏洞，导致 The DAO 被黑客攻击并转移走价值 6000 万美元的数字货币，最后不得不黯然落幕。这给以太坊生态系统带来了很大负面影响。此次事件之后，很多人对区块链的“去中心化”进行了反思：在挽回这个损失的过程中，原有的去中心化机制未能解决问题，最后还是通过“集中式”的方式，强制以太坊进行“硬分叉”完成交易回滚。

如果我们仔细研究中本聪的论文，就会发现其中只有 Peer-to-Peer (P2P)，而没有 Decentralized 一词。业界也正在逐渐对区块链并不一定去中心化形成共识。在 2016 年 6 月召开的 W3C 区块链标准会议上，以太坊的核心开发团队 EthCore 明确表示，不再使用 Decentralized 这个词，而是用 P2P、Secure、Serverless 这类纯技术性词语。

但是如果简单地宣称去中心化，会被误读成是在某种程度上存在着一种既想从事金融活动，又不愿意接受金融监管的倾向。

### 1.3 区块链技术平台

目前，全球有数个区块链技术平台，其中比特币 (Bitcoin)、以太坊 (Ethereum)、瑞波 (Ripple) 和 Linux 基金会的开源项目超级账本 (Hyperledger Fabric) 比较有代表性。

此外，还有比特股 (Bitshare)、恒星 (Stellar)、R3 Corda 等国外区块链技术以及国内一些公司研发的区块链应用平台。

#### 1.3.1 比特币

比特币 (Bitcoin) 是最早、全球使用最广泛的区块链技术，具有最去中心化、最多分布节点、最公平等特点。

比特币提出了一个不需要信用中介的数字货币系统，通过数字签名 (Digital Signatures) 使得在线支付能够直接由一方发起并支付给另外一方，中间不需要通过任何的金融机构。同时为了防止双重支付 (Double-Spending)，它提出了一种采用工作量证明机制的点对点网络来记录交易的公开信息，该网络通过随机散列 (Hashing) 对全部交易加上时间戳 (Time)，将它们合并入一个不断延伸的基于随机散列的工作量证明 (Proof of Work) 的链条作为交易记录，形成的交易记录将不可更改。只要诚实的节点能够控制绝大多数 CPU 的计算能力，



就能使攻击者难以改变交易记录。节点之间的工作，大部分是彼此独立的，只需要很少的协同。每个节点都不需要明确自己的身份，可以随时离开网络，若想重新加入网络也非常容易。节点通过自己的计算力进行投票，表决它们对有效区块的确认，它们不断延长有效的区块链来表达自己的确认，并拒绝在无效区块之后延长区块以表示拒绝。可以说，比特币包含了一个点对点数字货币系统所需要的全部规则和激励措施。

在比特币这种去中心化的公有区块链系统中，在相互间没有信任基础的前提下需要有一种完成点对点交易的共识机制。比特币发行的共识机制基于工作量证明算法（挖矿），使用过程基于点对点支付和全局记账，货币有效性基于追溯验证算法。“挖矿”过程就是把系统中没有记录的现有交易打包到区块里，通过系统提供的计算“挖矿”难度的随机数不断遍历，最先达到条件的会获得记录区块的权利。随后节点将该区块记录通过网络发布广播，全网其他节点在验证该区块满足条件，同时区块记录的交易符合规定后，分别把该区块记录的信息更新到自己节点的区块链上，从而形成全网账本的共识。

比特币区块链核心技术的框架采用 C++ 语言开发，共识算法采用 PoW 算法，工作量（挖矿）证明获得记账权，容错 50%，实现全网记账，公网性能 TPS 小于 7，开源地址为：<https://github.com/bitcoin/bitcoin>。比特币发行并运行到现在，说明了区块链技术在数字货币领域的可行性，但其并不能完全代表区块链技术，它只有唯一的数字资产（比特币），而且没有图灵完备的编程语言平台，允许开发人员建立更广泛的分布式账本系统应用。

比特币区块链推出的时间比较早也不够强大（如不支持智能合约）。现在当人们提到“区块链”时，往往已经与比特币网络没有直接联系了，除非特别指出是承载比特币交易系统的“比特币区块链”。但比特币仍是区块链最早也是截至目前数字货币方面最大并且在全球各地经常提及的应用。

在比特币源代码基础上，照搬或进行较小改动之后，还出现了一些区块链技术体系，其中包括一些山寨币，例如彩色币（染色币）等，还有以锚定比特币为基础的比特币侧链等。

### 1.3.2 以太坊

以太坊是一个通用的数字代币平台<sup>[3]</sup>，它通过一套图灵完备的脚本语言（Ethereum Virtual MachineCode, EVM 语言）建立应用，采用多种编程语言实现协议（编程并不需要使用 EVM 语言，而是使用类似 C 语言、Python、Lisp 等高级语言，再通过编译器转成 EVM 语言），采用 Go 语言编写的客户端作为默认客户端（即与以太坊网络交互的方法，支持其他多种语言的客户端）。以太坊 ETH 的开源地址：<https://github.com/ethereum>。

以太坊的核心目标是智能合约，它可以看作是一个以太坊系统里的自动代理人。它有一个自己的以太币地址，当用户向合约地址发送一笔交易后，该合约就会被激活，然后根据交易中的额外信息，合约运行自身的代码，最后返回一个结果，这个结果可能是从合约地址发出的另外一笔交易。需要指出的是，以太坊中的交易不只是发送以太币而已，它还可以嵌入

相当多的额外信息。如果一笔交易是发送给合约的，那么这些信息就非常重要，因为合约将根据这些信息来完成自身的业务逻辑。智能合约的引入对区块链 2.0 有着极大的推动作用，而作为早期推动智能合约的区块链平台，以太坊一度为广大区块链社区所看好。智能合约配合友好的界面和一些额外的小支持，可以让用户基于合约搭建各种千变万化的 DApp 应用，这样使得开发人员开发区块链应用的门槛大大降低。以太坊架构图如图 1-4 所示。

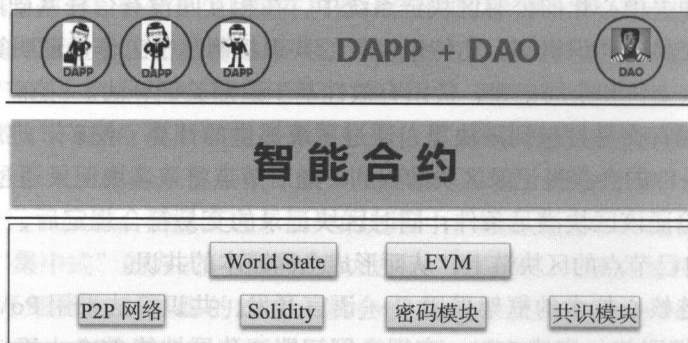


图 1-4 以太坊架构图

以太坊合并了很多对比特币用户来说十分熟悉的特征和技术，同时自己也进行了很多修正和创新。比特币区块链纯粹是一个关于交易的列表，而以太坊的基础单元是账户。以太坊区块链跟踪每个账户的状态，所有以太坊区块链上的状态转换都是账户之间价值和信息的转移。账户分为两类：

- 外部账户，由私人密码控制；
- 合约账户，由它们的合约编码控制，只能由外部账户“激活”。

对于大部分用户来说，两者的基本区别在于外部账户是由人类用户掌控——因为他们能够控制私钥，进而控制外部账户。而合约账户则是由内部编码管控。如果它们是被人类用户“控制”的，那也是因为程序设定了它们被具有特定地址的外部账户所控制，进而被持有私钥控制外部账户的人控制着。“智能合约”这个流行的术语指的是在合约账户中编码——交易发送给该账户时所运行的程序。用户可以在区块链中部署编码来创建新的合约。

以太坊迭代周期比较快，使得依赖于以太坊特别是以太坊公网的商业应用比较容易受到攻击。The DAO (The Distributed Autonomous Organization, 去中心化自治组织) 是区块链业界最大的众筹项目，它在短时间内就募集了价值 1.3 亿美元的数字货币。2016 年 6 月 17 日，由于其编写的智能合约存在重大缺陷，运行在以太坊公有链上的 The DAO 智能合约遭遇攻击<sup>①</sup>。导致 300 多万以太币资产被分离出 The DAO 资产池。The DAO 官方交流社区 DAOhub 称，在黑客风波结束及所有的以太币被解锁后，返还所有的以太币，并关闭 The DAO。

① The DAO 编写的智能合约中有一个 splitDAO 函数，攻击者通过此函数中的漏洞重复利用自己的资产不断从 The DAO 项目的资产池中分离资产给自己。

### 1.3.3 瑞波

瑞波 (Ripple) 是开放源码的点对点支付网络, 它可以轻松、廉价、安全、跨国界地进行转账。转账对象可以是互联网上的任何一个人, 无论他在世界的哪个地方; 被转账的可以是清算货币、虚拟货币、数字资产或任意一种有价值的资产。

瑞波使用的共识机制是 RPCA, 可以通过特殊节点的投票, 在很短的时间内对交易进行验证和确认。瑞波客户端不需要下载区块链, 它在普通节点上舍弃掉已经验证过的总账本链, 只保留最近已验证的总账本和一个指向历史总账本的链接, 因而同步和下载总账本的工作量很小。

作为世界上第一个开放的支付网络, 瑞波于 2015 年下半年对外公布了其 InterLedger 协议项目, 该项目的目标就是打造全球统一的支付标准, 创建统一的网络金融传输协议。通过瑞波支付网络可以转账任意一种货币, 包括美元、欧元、人民币、日元或者比特币, 简便、易行、快捷, 交易确认在几秒内完成, 交易费用几乎是零, 没有所谓的跨行异地以及跨国支付费用。而且由于是 P2P 软件, 所以没有任何个人、公司或政府操控瑞波, 任何人都可以创建一个账户。

目前, 瑞波币和以太坊之间正在争夺世界第二大加密货币的地位, 胜负尚未决出。2017 年早些时候, 瑞波网络已实现在 3.7s 内能够完成 7 万笔交易。瑞波公司为其瑞波共识账本 (RCL) 和互联账本协议 (ILP) 引入了新的功能 (托管和支付通道), 这提高了瑞波币 (XRP) 的交易吞吐量, 这些改进使得瑞波网络的可扩展性达到了 Visa 的级别, 也就是其交易吞吐量将可与 Visa 抗衡。

### 1.3.4 区块链商用平台: 超级账本

超级账本 (Hyperledger) 是 Linux 基金会的区块链项目, 致力于发展跨行业的商用区块链平台技术<sup>[4]</sup>。超级账本项目自创立伊始便吸引了众多行业的领头羊, 包括金融业、银行、互联网行业、运输业、制作业等。目前, 超级账本项目在全球拥有超过 100 个成员, 包括 Cisco、IBM、Intel、J.P.Morgan、荷兰银行、SWIFT 等。基于区块链技术、智能合约及其他相关技术, 超级账本项目致力于建立新一代的分布式账本交易应用平台, 从而在简化商业流程和法律事务的同时, 建立起商业信任、透明、审查能力。旗下的 Hyperledger Fabric 子项目是以 IBM 早期捐献出的 Open Blockchain 为主体搭建而成, 当时共向 HyperLedger 贡献了 44 000 行开源代码。

HyperLedger Fabric 是一个带有可插入各种功能模块架构的区块链实施方案, 目标是打造成由全社会共同维护的开源超级账本。开源地址: <https://github.com/hyperledger/fabric>。Fabric 的主要框架核心开发语言是 Go 语言, 其更适合于联盟链。早在 2016 年 IBM 宣布, 计划提供开源代码并持续向超级账本项目 (Hyperledger Project) 贡献区块链代码。IBM 将提供经过 IBM 测试与认证的超级账本区块链代码, 以及在多种技术平台上进行安装的方法, 以便开发者可以在容器 (Container) 内执行超级账本的代码, 并开始快速构建商品

溯源、贸易融资、信用证、供应链以及企业贷款等区块链网络。

可以说, Hyperledger 是对传统区块链模型的革新, 在某种程度上是允许创建授权和非授权的区块链。Hyperledger 还通过提供一个针对身份识别、可审计、隐私安全和健壮的模式, 使得缩短计算周期、提高规模效率和响应各个行业的应用需求成为可能。

利用超级账本平台, 用户可以轻松地搭建起企业级的区块链网络。在这个网络中, 每名成员都可以访问实时更新、加密过的账本, 并能查询及发起交易。一旦交易经过共识流程的验证, 它就会立即加入到网络中所有的账本中, 并且不能更改。交易结果迅速、私有、保密且易于审计。另外, 早期的超级账本还定义了协议规范: Open Blockchain Protocol Specification, 并以此建立了区块链平台 Hyperledger Fabric, 并可以用于一系列 B2B 和 B2C 交易相关的行业案例中。为了满足这些功能及要求, Hyperledger Fabric 的实现利用了下述概念:

- 智能合约 (smart contracts)
- 数字资产 (digital assets)
- 记录存储系统 (system of record repositories/stores)
- 基于共识的去中心化网络 (decentralized consensus-based network)
- 可插拔的共识算法及共识模型 (pluggable consensus algorithms/models)
- 加密安全机制 (cryptographic security)

这些概念和功能让 Hyperledger Fabric 架构结合了 3 个范畴: 成员管理、区块链和智能合约 (Chaincode)。这 3 个范畴是按逻辑划分的, 而不是在物理上对独立过程、地址空间或(虚拟)机器的组件分割。

### (1) 成员管理服务

成员管理提供了诸多服务, 包括身份管理、网络隐私、保密及审查。对于非准入型区块链, 参与者不需要提供认证许可, 所有的节点都平等地发起交易、验证交易及累积账本。也就是说, 在非准入型区块链中没有身份的区别。成员管理服务结合 PKI 技术和去中心化/共识, 将非准入型区块链转变为准入型区块链。在准入型区块链中, 参与者通过注册获取身份认证许可(注册证书), 并且通过参与类型区分类别。通过使用身份认证许可, 用户可以向交易认证中心(TCA)申请伪匿名认证许可。只有使用这样的许可信息(即交易证书)用户才可以发起交易。此外, 交易证书在区块链上永久存在, 审查人员可以以此追溯交易。

### (2) 区块链服务

通过建立在 HTTP/2 上的 P2P 协议, 区块链服务管理分布式账本。账本上的数据结构被高度优化, 从而支持对世界状态复制的高效哈希算法。此外, 在部署智能合约时, 还可以指定不同的共识算法, 如 PBFT、RAFT、PoW 和 PoS 等。

### (3) 智能合约

智能合约在 Fabric 中称为“链码”。链码服务为链码在验证节点上的执行提供了安全轻量级的沙箱。执行环境是一个“锁定”且安全的容器及一组签名镜像, 包含安全操作系统



和链码语言、运行时、SDK 层。链码语言包括 Go、Java 和 Node.js。此外，可以根据需求来启用其他语言。在网络中，验证节点与链码可以发出事件，应用程序可以监控并响应这些事件。目前已经预置了一些事件类型，链码还可以发出用户自定义事件。

### 1.3.5 区块链技术平台比较

各个区块链技术平台各有千秋，前面提到的共识机制、是否有智能合约功能、适用场景等都是进行比较的主要内容。

1) 智能合约：1995 年，跨领域法律学者和密码学家尼克·萨博（Nick Szabo）首次提出了“智能合约”（Smart contract）这一术语。当一个预先编好的条件被触发时，智能合约执行相应的合同条款。从本质上讲，这些智能合约的工作原理类似于其他计算机程序的 if-then 语句，智能合约只是以这种方式与真实世界的资产进行交互。以太坊和 HyperLedger Fabric 等以智能合约为核心的区块链越来越受到重视。

2) 适用场景：通常把区块链分为“公有链”（Public blockchain）、“私有链”（Private blockchain）和“联盟链”（Consortium blockchain）3 种。公有链对所有人开放，任何人都可以参与比特币，这是最典型的公有链；联盟链仅对特定的组织团体开放；私有链仅对单独的个人或实体开放。现在业内普遍认为联盟链介于公有链和私有链之间，可视为“部分去中心化”，公众可以查阅和交易，对于验证交易或发布智能合约等功能需要获得联盟许可。在非数字货币之外的场景中引入区块链技术时，使用哪种区块链，需要对诸多因素进行权衡决策。短期内，主流金融机构仍难以接纳公有链。共识算法的对比如表 1-1 所示。

表 1-1 共识算法对比

名称	共识算法	适合场景	开发语言	智能合约
比特币	PoW	公有链	C++	否
以太坊	PoW/PoS	公有链 / 联盟链	Go	是
瑞波	RPCA	公有链 / 联盟链	C++	否
HyperLedger Fabric	PBFT 为主	联盟链	Go	是

## 1.4 区块链的商用之道

区块链之所以称为一种“颠覆性”的新兴技术，因为尽管其成名于比特币，但未来区块链的用武之地将远远超过加密货币。区块链的分布式共享账本这一技术本质能够在商业网络中使更多的参与方得到更加广泛的参与，并为商业网络或行业业务带来更低的沟通或整合成本，以及更高的业务效率。可以预见，区块链作为一个独立的技术板块，会在商业领域中得到广泛应用。

### 1.4.1 区块链的 2.0 时代：商用区块链

自 2009 年比特币在交易领域迅速崛起以来，这种加密币受到了广泛关注，但也颇受争

议。不过比特币的底层技术——区块链，由于能够快速改进银行、供应链以及其他交易网络，在降低与业务运营相关的成本和风险的同时，带来创新和增长机会，所以是比较无争议的新兴技术模式，得到了商业界的鼎力支持。

传统的商业业务模式存在的问题是很难在一个互信的网络中监视跨机构的交易执行：每个参与方都有自己的账本，在交易发生时各自更改；协同各方会导致额外的工作及中介等附加成本；由于业务条件的不同，“合同”重复分散在各个参与方，造成整体业务流程的低有效性；整个业务网络依赖于一个或几个中心系统，整个商业网络十分脆弱。

而区块链提供了共享、复制、授权的账本这样一个解决方案。区块链架构带来了以下改变：

- 1) 区块链架构使每一个商业网络的参与方都具有一个共享账本，当交易发生时，通过点对点的复制更改所有账本；
- 2) 使用密码算法确保网络上的参与者仅仅可以看到和他们相关的账本内容，交易是安全的、授权的和验证的；
- 3) 区块链也将与资产转移交易相关的合同条款嵌入交易数据库中以做到在满足商务条件下交易才发生；
- 4) 网络参与者基于共识机制或类似的机制来保证交易是共同验证的，商业网络满足政府监管、合规及审计。

总体而言，区块链在提高业务效率和简化流程上确实具有优势。

当前国内外区块链产业生态发展迅猛，产业链层次逐渐清晰，无论是底层基础架构和平台，还是细分产业板块的区块链应用，以及风险资本投资都已初具规模。综合来看，全球区块链在商业行业发展具有三大趋势。

1) **从比特币向更丰富的应用场景发展。**区块链 2.0 把之前“区块链就是比特币”的意义向前推进了一大步，区块链也不再是比特币的专有技术和代名词，而是在更加广泛的应用场景中，成为资产流转的价值表述。区块链当下不再依赖数字货币或资产这一类单一场景，而是发展到支付汇兑、电子商务、移动社交、众筹、慈善、互助保险等面向终端用户的应用，以及数字资产、IP 版权和交易、金融清算和结算、商品溯源等企业级应用领域。

2) **全球区块链生态日益丰富，参与方开始出现明显的产业分工。**从全球视角来看，随着参与者越来越多，区块链形成了不同的技术平台、行业以及发展路径的产业生态。对全球的区块链从业者来说，更看重区块链作为未来金融科技（FinTech）的一个领域从而提前布局，大胆尝试实践区块链在金融及其他行业中的各类业务场景。同时，高科技龙头企业也希望在区块链技术框架的建立上尽早发力，通过支持全球开源社区建立更扎实的底层区块链平台和更广泛的应用场景。区块链的热潮带动了更多创业者的热情，众多初创公司如雨后春笋般应运而生。从行业角度来看，区块链初创公司覆盖了银行和保险服务、供应链、医疗、物联网、外贸等众多行业，可谓是百花齐放。

3) **全球投资正在快速注入，重点关注企业级应用落地。**区块链项目融资正呈现井喷式增长，从 2012 年到 2015 年，区块链领域吸引的风险投资从 200 万美元增长到 4.69 亿美

元,增长超过了200倍,累计投资已达10亿美元左右。2016年仅在金融领域,区块链技术投资额就占整体投资的七成以上。从全球的投资情况来看,由于越来越多的行业已经开始实践区块链,所以使得更多投资人开始关注行业内的区块链应用场景,投资趋于理性,但更注重利用投资者自身资源帮助投资标的进行深度的行业孵化。

总之,市场、行业、投资等多方对于商用区块链的发展诉求十分强烈,作为“颠覆性创新”技术的区块链前景光明。

### 1.4.2 超级账本: 商用区块链的“第五元素”

企业级区块链四大平台要素包括:共享账本,共识、隐私和保密、智能合约。此外,还有第五要素,即商业网络。企业级的区块链一定是围绕业务场景展开的,因此在第五元素商业网络当中需要包含市场参与者的对等架构以及伙伴间的一个共识协议。

目前,以比特币为代表的公有链有一些加密货币之外的新型应用,但是却无法克服自身固有的一些问题,例如交易效率低,区块没有最终确定性(finality)等,而且是由极客主导的,不符合商业主流趋势。为了克服上述不足,满足大多数商业应用的要求,设计开发适合商用的区块链平台迫在眉睫。

企业级商用区块链网络比较适合使用联盟链和许可制。这样在一个限定的范围内,只有授权的节点和用户才能参与到交易和智能合约的执行中来,而任何匿名节点或非授权用户均被拒绝服务。从团体联盟的角度来看,这增加了区块链网络的安全可靠。当前,在欧美主流的区块链应用大部分是行业链或者是联盟链,也就是某一个行业的上下游,或者核心企业联合起来,一起构建的半公开化的区块链。从这个角度讲,超级账本具备成为未来最主要商用区块链技术平台的潜力,值得技术开发人员花时间和精力进行学习和研究。

由于超级账本有个重要的设计原则是按照“用例驱动”(use case driven)的方式来实现的,所有功能都应该有对应的用例需求,因此学习研究的过程并不一定十分辛苦。此外,鉴于超级账本是个通用型框架,无法预先确定将来所有的应用场景,因此,定义出部分典型的用例,可使超级账本先满足这部分有代表性的区块链应用需求,然后再用可替换模块满足其他需求。

### 1.4.3 区块链的商业应用场景

区块链的商业应用才刚刚起步,一般都将金融业应用作为切入口,很多其他领域的應用还在探索或试水阶段。最重要的是,不能为了技术而技术,为了区块链而区块链。商用区块链技术要解决企业的痛点,为客户创造新的价值。可喜的是,在金融和金融以外的各个细分领域,区块链都在加速落地。以下为一些应用实例和构想。

1) 金融领域。21世纪是金融的“大航海时代”,对银行、保险、清算、股权登记交易、信用评级、公证等领域,既需要绝对的可信任,也需要隐私保密,所以特别适合区块链应用。举例来说,金融行业关心的资产分布式管存,可以把资产(如证券等)数据存放在区块

链网络中，资产利益相关人可以直接访问资产数据，而无须经过传统的中间人，可大幅提高效率和节约成本。区块链股权登记和交易平台脱胎于加密货币交易所，也是比较合适及容易实现的应用。

2) **产业互联网领域。**供应链溯源和共享经济可以应用区块链。在供应链中，所有的参与者都通过区块链记录、追踪和共享各种数据，这些数据记录存储在区块链里面并贯穿货物的生产、运输和销售等环节，从而提供深度回溯、查询等核心功能，实现信息公开透明，出了问题可以依此来追责。附加值较高的食品、药品和疫苗、零部件生产检测结果等都可以使用区块链。

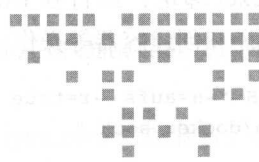
例如，现在市场上号称是北大荒地区生产的大米，特别是五常大米，是当地实际产量的很多倍，造成良莠不齐。消费者希望花比较高的价钱购买真正的北大荒大米，却苦于无法分辨哪些大米是真的。此处可以体现区块链在供应链溯源上的价值，就是利用区块链中数据记录的真实性或者有效性。如果在原产地和各个流通环节中设置的传感设备在区块链上签名盖戳，一旦进入到区块链里面，每个人的签名就不能抵赖了。含有被区块链标记的时间戳、地理戳、品质戳的放心粮从源头上杜绝了各个环节作弊的动机，这样市场上才能销售与当地产量相匹配的大米。

3) **传统行业的转型创新。**区块链的应用绝不仅局限于金融和互联网等前沿领域，还可以与能源、零售、电商、房地产等传统领域接轨，因此区块链不是个摆设。例如，高盛公司就提出，对于资产所有权需经过谨慎识别的房地产交易，如果能够利用区块链技术建立安全、共享的所有权数据库，那么房产交易纠纷和交易成本将大大缩小。

4) **FinTech2.0 的三驾马车：区块链、认知物联网和人工智能。**未来，世界将进入人工智能、认知物联网和区块链三足鼎立的时期，如果能将三者有机结合将创造巨大的价值。例如，如果将闲置或未充分利用的资产（如汽车、仓库、医疗设备等）接入物联网，那么区块链技术可以帮助互不相识的这些资产的所有者进行资产使用的交易谈判。在共享经济的模式下，最需要解决的就是陌生人之间的信任问题，即资源的提供方和资源的租用者，如何在缺乏信任的基础上安全地完成交易。分布式区块链将是一种全新的去信任方式，不使用任何中间平台，达到各方参与者可靠交易的目的。这有点类似于分时用车和分时用房，这将引爆以前隐藏在深处的过剩资产容量。

## 1.5 本章小结

区块链作为当下最流行的技术之一，从提出概念原型到造就价格惊人的比特币只用了不到十年的时间，而且它的用途还可以扩展到商业的多个领域。本章介绍了区块链的概念，包括其本质、工作原理、技术特点、层次模型、共识算法等，还特别纠正了“区块链一定是去中心化的”错误观念。我们还对比特币、以太坊、瑞波、超级账本等技术进行了简要介绍和比较。现在我们大致了解了区块链的世界，下面可以具体学习超级账本了。



## 第 2 章 Chapter 2

# 超级账本初体验

本章先简单介绍一下 Hyperledger Fabric 1.0 的环境搭建，快速地体验一下超级账本的功能。本书所有的内容都是基于 Hyperledger Fabric 1.0 的，在后面章节中我们偶尔也会用到“超级账本”这个词，指的也是超级账本的 Hyperledger Fabric 1.0 项目。

## 2.1 基础环境安装

Hyperledger Fabric 1.0 依赖 Docker 执行智能合约，需要先安装 Docker 和 Docker Compose 的运行环境。

### 2.1.1 Docker 的安装和使用

Docker 支持 Linux、Mac、Windows 等多个平台，安装文档参考：<https://docs.docker.com/engine/installation>。

#### 1. 在 Linux 环境下 Docker 的安装

Ubuntu、Debian、CentOS 等 Linux 系统，可以通过 Docker 官方提供的脚本进行安装：

```
curl -sSL https://get.docker.com | sh
```

然后把用户加入到 docker 组，非 root 用户 USER 可以执行 docker 命令（可能需要重新登录生效）：

```
sudo usermod -aG docker $USER
```

如果是 Ubuntu 或者 Debian 操作系统，修改 Docker 的配置文件 /etc/default/docker，增



加 Docker 的 socket 绑定, 运行在 Docker 中的进程才能通过映射的 socket 调用 Docker 的 API 执行镜像编译和创建容器等操作。

```
DOCKER_OPTS="-s=aufs -r=true --api-cors-header='*' -H tcp://0.0.0.0:2375 -H
unix:///var/run/docker.sock "
```

接着, 重启 Docker 服务让配置生效:

```
sudo service docker start
```

CentOS 系统采用 Systemd 进行系统和服务管理, 配置文件的修改方法是不一样的。CentOS 系统下 Docker 的配置文件是 /etc/sysconfig/docker, 同样要修改 DOCKER\_OPTS 选项。还需要修改 /usr/lib/systemd/system/docker.service 文件, 在 [Service] 的 ExecStart= 下面增加一行 \$DOCKER\_OPTS, 如下所示:

```
[Service]
Type=notify
NotifyAccess=all
EnvironmentFile=-/etc/sysconfig/docker
EnvironmentFile=-/etc/sysconfig/docker-storage
EnvironmentFile=-/etc/sysconfig/docker-network
Environment=GOTRACEBACK=crash
Environment=DOCKER_HTTP_HOST_COMPAT=1
Environment=PATH=/usr/libexec/docker:/usr/bin:/usr/sbin
ExecStart=/usr/bin/dockerd-current \
    --add-runtime docker-runc=/usr/libexec/docker/docker-runc-current \
    --default-runtime=docker-runc \
    --exec-opt native.cgroupdriver=systemd \
    --userland-proxy-path=/usr/libexec/docker/docker-proxy-current \
    $DOCKER_OPTS \
    $OPTIONS \
    $DOCKER_STORAGE_OPTIONS \
    $DOCKER_NETWORK_OPTIONS \
    $ADD_REGISTRY \
    $BLOCK_REGISTRY \
    $INSECURE_REGISTRY
```

重启服务让配置生效:

```
systemctl daemon-reload
systemctl restart docker.service
```

## 2. 其他环境下 Docker 的安装

Windows 和 Mac 都提供了安装包, 直接下载即可安装:

- ☐ Docker for Mac: <https://download.docker.com/mac/stable/Docker.dmg>
- ☐ Docker for Windows: <https://download.docker.com/win/stable/InstallDocker.msi>



3. Docker 国内镜像仓库

国外的镜像下载较慢，可以设置国内的镜像，阿里云和 DaoCloud 都提供镜像加速的服务，需要登录注册才能使用。

- ❑ 阿里云：登录容器 Hub 服务 <https://cr.console.aliyun.com> 的控制台，左侧的加速器帮助页面会显示为你独立分配的加速地址。
- ❑ DaoCloud：在 <https://www.daocloud.io> 进行注册登录，然后点击加速器，就可以获取加速器的相关配置。

修改 Docker 镜像仓库的办法是在 DOCKER\_OPTS 里增加 registry-mirror 参数，比如：

```
DOCKER_OPTS="-s=aufs -r=true --api-cors-header='*' -H tcp://0.0.0.0:2375  
-H unix:///var/run/docker.sock  
--registry-mirror=http://069f616f.m.daocloud.io"
```

重启 Docker 服务就可以使用镜像加速了。

Docker 在 Windows 和 Mac 中的版本可以在图形界面添加镜像仓库。

4. Docker 常用命令

Docker 常用命令如表 2-1 所示。

表 2-1 Docker 常用命令

命令	举例	说明
docker images	docker images	查看主机上的镜像文件列表
docker pull	docker pull hyperledger/fabric-peer	从镜像仓库中下载镜像文件
docker tag	docker tag hyperledger/fabric-tools:x86_64-1.0.0 hyperledger/fabric-tools:latest	给镜像文件打标签，x86_64-1.0.0 标记为 latest
docker run	docker run -it --name cli ubuntu /bin/bash	从镜像中启动容器，其中：cli 是容器名称，ubuntu 是镜像名称，-it 是以交互方式启动，/bin/bash 是启动容器时执行的命令
docker logs	docker logs -f cli	查看容器日志
docker ps	docker ps -a	查看主机上的容器，其中：参数 -a 会显示已经停止的容器
docker port	docker port peer0.org1.example.com	查看容器映射的端口
docker rm	docker rm cli	删除容器，其中：cli 为容器名称或者 id
docker --help	docker --help	查看帮助文档

更多的命令请查看帮助文档和在线文档：<https://docs.docker.com/engine/reference/commandline/docker>。

2.1.2 Docker Compose 的安装和使用

Docker Compose 能够在一个主机上创建出相互隔离的网络，通过命令行管理多个 Docker 容器，快速启动、停止和更新容器。

## 1. Docker Compose 的安装

Docker 在 Windows 和 Mac 中都已经集成了 Docker Compose 工具，不需要单独安装。在 Linux 系统下有多种安装方法，如下所示：

### (1) 通过 pip 进行安装

```
sudo apt install python-pip
sudo pip install docker-compose
```

### (2) 直接下载文件

```
curl -L
https://github.com/docker/compose/releases/download/1.17.1/docker-compose-`uname -s`-
`uname -m` -o /usr/local/bin/docker-composechmod +x /usr/local/bin/docker-compose
```

## 2. Docker Compose 的配置文件

Compose 采用 YAML 文件定义 Docker 容器之间的依赖，设置环境变量和文件的持久化。我们看一个配置文件 examples/e2e\_cli/base/docker-compose-base.yaml 的节选：

```
version: '2'
```

```
services:
```

```
  orderer.example.com:
```

```
    container_name: orderer.example.com
```

```
    image: hyperledger/fabric-orderer
```

```
    environment:
```

```
      - ORDERER_GENERAL_LOGLEVEL=debug
      - ORDERER_GENERAL_LISTENADDRESS=0.0.0.0
      - ORDERER_GENERAL_GENESISMETHOD=file
```

```
      - ORDERER_GENERAL_GENESISFILE=/var/hyperledger/orderer/orderer.genesis.block
```

```
      - ORDERER_GENERAL_LOCALMSPID=OrdererMSP
```

```
      - ORDERER_GENERAL_LOCALMSPDIR=/var/hyperledger/orderer/msp
```

```
      # enabled TLS
```

```
      - ORDERER_GENERAL_TLS_ENABLED=true
```

```
      - ORDERER_GENERAL_TLS_PRIVATEKEY=/var/hyperledger/orderer/tls/server.key
```

```
      - ORDERER_GENERAL_TLS_CERTIFICATE=/var/hyperledger/orderer/tls/server.crt
```

```
      - ORDERER_GENERAL_TLS_ROOTCAS=[/var/hyperledger/orderer/tls/ca.crt]
```

```
    working_dir: /opt/gopath/src/github.com/hyperledger/fabric
```

```
    command: orderer
```

```
    volumes:
```

```
      - ../channel-artifacts/genesis.block:/var/hyperledger/orderer/orderer.genesis.block
```

```
      - ../crypto-config/ordererOrganizations/example.com/orderers/orderer.example.com/
msp:/var/hyperledger/orderer/msp
```

```
      - ../crypto-config/ordererOrganizations/example.com/orderers/orderer.example.com/
tls:/var/hyperledger/orderer/tls
```

```
    ports:
```

```
      - 7050:7050
```

```

peer0.org1.example.com:
  container_name: peer0.org1.example.com
  extends:
    file: peer-base.yaml
    service: peer-base
  environment:
    - CORE_PEER_ID=peer0.org1.example.com
    - CORE_PEER_ADDRESS=peer0.org1.example.com:7051
    - CORE_PEER_CHAINCODELISTENADDRESS=peer0.org1.example.com:7052
    - CORE_PEER_GOSSIP_EXTERNALENDPOINT=peer0.org1.example.com:7051
    - CORE_PEER_LOCALMSPID=Org1MSP
  volumes:
    - /var/run/:/host/var/run/
    - ../crypto-config/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/msp:/etc/hyperledger/fabric/msp
    - ../crypto-config/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls:/etc/hyperledger/fabric/tls
  ports:
    - 7051:7051
    - 7052:7052
    - 7053:7053

```

在这个节选的配置文件中，一共定义了1个排序服务节点 `orderer.example.com` 和1个Peer节点 `peer0.org1.example.com`。Docker Compose 目前有3个版本，这个配置文件采用的 version 2 的语法，配置文件的解释如表 2-2 所示。

表 2-2 配置文件的解释

选项	举例	说明
version	version: '2'	采用 version 2 的语法
services		定义服务列表
orderer.example.com	根据服务名称自定义	自定义的服务名称，需要保持唯一
container_name	container_name: orderer.example.com	容器名称
image	image: hyperledger/fabric-orderer	容器使用的镜像文件
environment	- CORE_PEER_LOCALMSPID=Org1MSP	传递给容器的环境变量
working_dir	working_dir: /opt/gopath/src/github.com/hyperledger/fabric	容器启动的工作目录
command	command: orderer	容器启动命令
volumes	- /var/run:/host/var/run/	宿主机和容器之间的目录映射
ports	- 7050:7050	宿主机和容器之间的端口映射
extends	file: common.yml	服务扩展，基于 common.yml 文件
extends	service: peer-base	服务扩展，基础服务是 peer-base

更多不同版本的配置文件说明请参考在线帮助文档：<https://docs.docker.com/compose/compose-file>。

3. Docker Compose 的常用命令

Docker Compose 的常用命令如表 2-3 所示。

表 2-3 Docker Compose 的常用命令

命令	举例	说明
docker-compose up	docker-compose -f docker-compose-cli.yaml up -d	根据配置文件 docker-compose-cli.yaml 启动容器，其中：-f 指定配置文件的名称，-d 设置以后台方式运行
docker-compose down	docker-compose -f docker-compose-cli.yaml down	停止配置文件 docker-compose-cli.yaml 的容器
docker-compose pull	docker-compose -f docker-compose-cli.yaml pull	批量下载所需的镜像文件

更多的命令查看帮助文档和在线文档：<https://docs.docker.com/compose/reference>。

2.1.3 下载超级账本源代码

超级账本的源代码都托管在 <https://gerrit.hyperledger.org/r/#/admin/projects/> 下面，并在 <https://github.com/hyperledger> 上提供只读代码。最好的方式是直接通过 git 下载：

```
git clone https://github.com/hyperledger/fabric.git
```

也可以打包下载文件后解压：<https://github.com/hyperledger/fabric/archive/release.zip>。

2.2 超级账本部署调用

先最小化地体验一下超级账本的环境，更详细的部署流程参考第 11 章。

2.2.1 下载 Docker 镜像文件

超级账本源码 scripts 目录下有多个下载镜像的脚本，我们可以修改权限以后直接运行：

```
# 进入 fabric/scripts 目录
chmod +x bootstrap-1.0.0.sh
# MacOS 系统执行如下命令（不下载二进制文件）
sed -i '' 's/curl/#curl/g' bootstrap-1.0.0.sh
# 其他系统执行如下命令（不下载二进制文件）
sed -i 's/curl/#curl/g' bootstrap-1.0.0.sh
# 直接下载 Docker 镜像文件
./bootstrap-1.0.0.sh
```

根据网络情况，可能需要等待一段时间。下面是下载的 Docker 镜像文件：

```
localhost:dive-into-fabric clarity$ docker images
```

REPOSITORY	TAG	IMAGE ID	SIZE
hyperledger/fabric-tools	latest	0403fd1c72c7	1.32GB
hyperledger/fabric-tools	x86_64-1.0.0	0403fd1c72c7	1.32GB
hyperledger/fabric-couchdb	latest	2fbdbf3ab945	1.48GB

hyperledger/fabric-couchdb	x86_64-1.0.0	2fbdbf3ab945	1.48GB
hyperledger/fabric-kafka	latest	dbd3f94de4b5	1.3GB
hyperledger/fabric-kafka	x86_64-1.0.0	dbd3f94de4b5	1.3GB
hyperledger/fabric-zookeeper	latest	e545dbf1c6af	1.31GB
hyperledger/fabric-zookeeper	x86_64-1.0.0	e545dbf1c6af	1.31GB
hyperledger/fabric-orderer	latest	e317ca5638ba	179MB
hyperledger/fabric-orderer	x86_64-1.0.0	e317ca5638ba	179MB
hyperledger/fabric-peer	latest	6830dcd7b9b5	182MB
hyperledger/fabric-peer	x86_64-1.0.0	6830dcd7b9b5	182MB
hyperledger/fabric-javaenv	latest	8948126f0935	1.42GB
hyperledger/fabric-javaenv	x86_64-1.0.0	8948126f0935	1.42GB
hyperledger/fabric-ccenv	latest	7182c260a5ca	1.29GB
hyperledger/fabric-ccenv	x86_64-1.0.0	7182c260a5ca	1.29GB
hyperledger/fabric-ca	latest	a15c59ecda5b	238MB
hyperledger/fabric-ca	x86_64-1.0.0	a15c59ecda5b	238MB

REPOSITORY 代表的是镜像的仓库名称，每个仓库下面都有打了不同 TAG 的标签名称，代表不同的版本。通常最少有两个标签，一个是 latest；另外一个的命名规则是“主机 CPU 类型 - 超级账本主版本号 - snapshot - 代码库版本号”，其中主机 CPU 类型为 x86\_64，说明是 Intel 的 64 位 CPU，超级账本的主版本为 1.0.0，snapshot 是固定名称，代码库版本号为 58cde93，它是 git 代码库最近一次提交版本号的前 7 位。snapshot 和代码库版本号只有通过本地编译的时候才会出现。每次 make docker 的时候都会检查是否有文件改动，如果有变化的文件，则会重新构建，生成新的镜像再标记成 latest。镜像文件详细的解释请参考第 11 章的相关内容。

2.2.2 部署超级账本网络

运行超级账本需要设置较多的初始化配置，我们先绕开初始化过程，用 fabric-samples 工程中已经生成的配置文件来体验部署安装的过程：

```
git clone https://github.com/hyperledger/fabric-samples.git
```

进入 basic-network 目录，利用 docker-compose 启动容器：

```
cd fabric-samples/basic-network
docker-compose -f docker-compose.yml up -d
```

查看已经启动的容器（输出进行了删减）：

localhost:basic-network clarity\$ docker ps		
CONTAINER ID	IMAGE	NAMES
efddfbf4fc0a	hyperledger/fabric-peer:x86_64-1.0.0	peer0.org1.example.com
606d13c1e7a2	hyperledger/fabric-couchdb:x86_64-1.0.0	couchdb
d8c870db8634	hyperledger/fabric-ca:x86_64-1.0.0	ca.example.com
c6f25a5e6fd6	hyperledger/fabric-tools:x86_64-1.0.0	cli
a5f6331c5bc5	hyperledger/fabric-orderer:x86_64-1.0.0	orderer.example.com

切换到管理员用户再创建通道和加入通道：

```
# 切换环境到管理员用户的 MSP, 进入 Peer 节点容器 peer0.org1.example.com
docker exec -it -e "CORE_PEER_MSPCONFIGPATH=/etc/hyperledger/msp/users/Admin@org1.example.com/msp" peer0.org1.example.com bash
# 创建通道
peer channel create -o orderer.example.com:7050 -c mychannel -f /etc/hyperledger/configtx/channel.tx
# 加入通道
peer channel join -b mychannel.block
# 退出 Peer 节点容器 peer0.org1.example.com
exit
# 退出 Peer 节点容器 peer0.org1.example.com, 进入 cli 容器安装链码和实例化:
# 进入 cli 容器
docker exec -it cli /bin/bash
# 给 Peer 节点 peer0.org1.example.com 安装链码
peer chaincode install -n mycc -v v0 -p github.com/chaincode_example02
# 实例化链码
peer chaincode instantiate -o orderer.example.com:7050 -C mychannel -n mycc -v v0 -c '{"Args":["init","a","100","b","200"]}'
```

## 2.2.3 链码调用和查询

链码实例化以后, 可以查询初始值, 同样是在 cli 容器里执行下面的操作:

```
peer chaincode query -C mychannel -n mycc -v v0 -c '{"Args":["query","a"]}'
```

查询结果显示为 Query Result: 100, 详细信息如下:

```
2017-08-09 14:47:05.853 UTC [msp] GetLocalMSP -> DEBU 001 Returning existing local MSP
2017-08-09 14:47:05.853 UTC [msp] GetDefaultSigningIdentity -> DEBU 002 Obtaining default signing identity
2017-08-09 14:47:05.853 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 003 Using default escc
2017-08-09 14:47:05.854 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 004 Using default vscc
2017-08-09 14:47:05.854 UTC [msp/identity] Sign -> DEBU 005 Sign: plaintext: 0A91070A6708031A0C08A9A694D00510...6D7963631A0A0A0571756572790A0161
2017-08-09 14:47:05.854 UTC [msp/identity] Sign -> DEBU 006 Sign: digest: E18FC97C13D550C5E3349AAD49523A6D7C71B4E51C219CD9A8799DEF54FFFE66
Query Result: 100
2017-08-09 14:47:05.886 UTC [main] main -> INFO 007 Exiting.....
```

调用链码, 从 “a” 转移 10 到 “b”:

```
peer chaincode invoke -C mychannel -n mycc -v v0 -c '{"Args":["invoke","a","b","10"]}'
```

显示调用成功的结果:

```
2017-08-09 14:49:46.018 UTC [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 0cb
```



```
Chaincode invoke successful. result: status:200
```

再次查询“a”和“b”的值:

```
peer chaincode query -C mychannel -n mycc -v v0 -c '{"Args":["query","a"]}'
peer chaincode query -C mychannel -n mycc -v v0 -c '{"Args":["query","b"]}'
```

查询结果显示“a”的值为 Query Result: 90, “b”的值为 Query Result: 210。

## 2.2.4 常见错误

### 1. 请求调用者权限不足

调用的时候设置了错误的 MSP, 比如需要管理员才能执行创建通道的操作, 但是设置了普通的成员 MSP, 会出现 Error: Got unexpected status: BAD\_REQUEST 的错误:

```
2017-08-09 14:49:04.652 UTC [msp] GetLocalMSP -> DEBU 001 Returning existing
local MSP
2017-08-09 14:49:04.652 UTC [msp] GetDefaultSigningIdentity -> DEBU 002 Obtaining
default signing identity
2017-08-09 14:49:04.654 UTC [channelCmd] InitCmdFactory -> INFO 003 Endorser and
orderer connections initialized
2017-08-09 14:49:04.656 UTC [msp] GetLocalMSP -> DEBU 004 Returning existing
local MSP
2017-08-09 14:49:04.656 UTC [msp] GetDefaultSigningIdentity -> DEBU 005 Obtaining
default signing identity
2017-08-09 14:49:04.657 UTC [msp] GetLocalMSP -> DEBU 006 Returning existing
local MSP
2017-08-09 14:49:04.657 UTC [msp] GetDefaultSigningIdentity -> DEBU 007 Obtaining
default signing identity
2017-08-09 14:49:04.657 UTC [msp/identity] Sign -> DEBU 008 Sign: plaintext: 0A8
8060A074F7267314D535012FC052D...53616D706C65436F6E736F727469756D
2017-08-09 14:49:04.657 UTC [msp/identity] Sign -> DEBU 009 Sign: digest: F77320
AE89B131CE75A858A4A450CF0F35301DA62FE1DE465CAEF4439F6FC520
2017-08-09 14:49:04.657 UTC [msp] GetLocalMSP -> DEBU 00a Returning existing
local MSP
2017-08-09 14:49:04.657 UTC [msp] GetDefaultSigningIdentity -> DEBU 00b Obtaining
default signing identity
2017-08-09 14:49:04.657 UTC [msp] GetLocalMSP -> DEBU 00c Returning existing
local MSP
2017-08-09 14:49:04.657 UTC [msp] GetDefaultSigningIdentity -> DEBU 00d Obtaining
default signing identity
2017-08-09 14:49:04.657 UTC [msp/identity] Sign -> DEBU 00e Sign: plaintext: 0AB
F060A1508021A0608E0D591D00522...A38A58EED7B94AC4CB800B86F0A5EF03
2017-08-09 14:49:04.658 UTC [msp/identity] Sign -> DEBU 00f Sign: digest: 475A33
426FA36D50F090AAF7C3AAAB2BF34339191BA74D4A60BF13460B241329
Error: Got unexpected status: BAD_REQUEST
Usage:
peer channel create [flags]
```

## Flags:

- c, --channelID string In case of a newChain command, the channel ID to create.
- f, --file string Configuration transaction file generated by a tool such as configtxgen for submitting to orderer
- t, --timeout int Channel creation timeout (default 5)

## Global Flags:

- cafile string Path to file containing PEM-encoded trusted certificate(s) for the ordering endpoint
- logging-level string Default logging level and overrides, see core.yaml for full syntax
- o, --orderer string Ordering service endpoint
- test.coverprofile string Done (default "coverage.cov")
- tls Use TLS when communicating with the orderer endpoint
- v, --version Display current version of fabric peer server

## 2. 传递错误的通道名称

比如通道名称是 mychannel, 传递参数的时候写成了错误的 myc:

```
peer chaincode query -C myc -n mycc -v v0 -c '{"Args":["query","a"]}'
```

会出现如下错误:

```
2017-08-09 14:40:36.703 UTC [msp] GetLocalMSP -> DEBU 001 Returning existing local MSP
2017-08-09 14:40:36.703 UTC [msp] GetDefaultSigningIdentity -> DEBU 002 Obtaining default signing identity
2017-08-09 14:40:36.706 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 003 Using default escc
2017-08-09 14:40:36.706 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 004 Using default vscc
2017-08-09 14:40:36.707 UTC [msp/identity] Sign -> DEBU 005 Sign: plaintext: 0A91070A6708031A0C08A4A394D00510...30300A000A04657363630A0476736363
2017-08-09 14:40:36.707 UTC [msp/identity] Sign -> DEBU 006 Sign: digest: F085E189A0765713209DD8802DDF57B054EBCD294305A500F56BC34CB3D2E577
Error: Error endorsing chaincode: rpc error: code = Unknown desc = chaincode error (status: 500, message: chaincode exists mycc)
Usage:
    peer chaincode instantiate [flags]
```

## Flags:

- C, --channelID string The channel on which this command should be executed (default "testchainid")
- c, --ctor string Constructor message for the chaincode in JSON format (default "{}")
- E, --escc string The name of the endorsement system chaincode to be used for this chaincode
- l, --lang string Language the chaincode is written in (default "golang")
- n, --name string Name of the chaincode

```

-P, --policy string      The endorsement policy associated to this chaincode
-v, --version string     Version of the chaincode specified in install/
                           instantiate/upgrade commands
-V, --vscc string        The name of the verification system chaincode to be
                           used for this chaincode

```

Global Flags:

```

--cafile string          Path to file containing PEM-encoded trusted
                           certificate(s) for the ordering endpoint
--logging-level string    Default logging level and overrides, see core.
                           yaml for full syntax
-o, --orderer string      Ordering service endpoint
--test.coverprofile string Done (default "coverage.cov")
--tls                    Use TLS when communicating with the orderer endpoint

```

## 2.3 节点的配置参数传递规则

在 docker-compose.yml 文件中，我们可以看到有 ORDERER\_GENERAL\_LEDGERTYPE=ram 的设置，这是传递给节点的参数。给节点传递参数的方法有多种方式：环境变量、配置文件、动态环境变量、默认值。

程序在启动的时候会读取配置文件和环境变量的值，分别保存到不同变量缓存起来，在程序需要获取某个变量值的时候，不同传递方法的参数读取流程图如图 2-1 所示。

从图 2-1 中可以看到，如果配置了自动从环境变量获取参数的值，那么每次都实时地从环境变量中获取，否则依次读取程序启动时从环境变量、配置文件中读取后缓存到内存中的值，优先获取到的值作为返回值。如果都没有获取到，则返回空，交给程序进行处理，程序可能会以默认值运行，也可能会报错停止运行，这跟业务逻辑有关系。所以，只有在设置了自动从环境变量中获取参数的情况下，才能在运行时通过修改环境变量改变参数的值。

每个环境变量的名称都有一个前缀，每个模块都是单独设置的，比如 ORDERER\_GENERAL\_LEDGERTYPE 的前缀是 ORDERER，通常每个模块的前缀是不一样的。比如这里的 ORDERER 代表的是排序服务节点，Peer 节点的变量名称前缀是 CORE。环境变量名称是以 “\_” 作为分隔符的，代表一种层级，是和配置文件一一对应的，比如 ORDERER\_GENERAL\_LEDGERTYPE 对应 orderer.yaml 配置文件的 General.LedgerType。环境变量和配置文件的变量名称都是不区分大小写的，内部会统一转换成小写的变量名称进行处理。

配置文件路径优先读取环境变量设置的路径，排序服务节点和 Peer 节点都是相同的环境变量。如果没有设置环境变量，则默认是应用程序所在的目录，然后环境变量 GOPATH 路径对应到模块代码工程下的目录，比如排序服务节点的目录是 \$GOPATH/src/github.com/hyperledger/fabric/orderer，Peer 节点的目录是 \$GOPATH/src/github.com/hyperledger/fabric/peer。

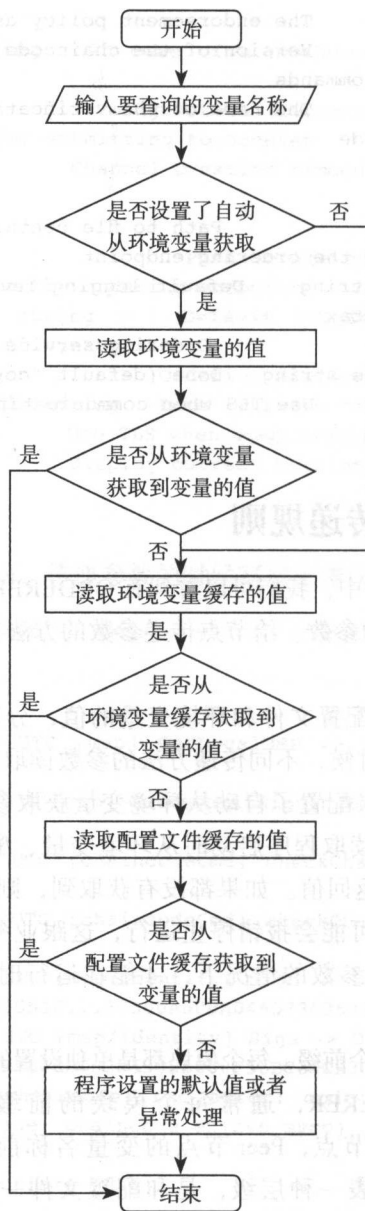


图 2-1 节点配置参数的传递规则

一般情况下，配置文件的名称设置成模块前缀的小写，比如排序服务节点的配置文件名称是 `orderer`，Peer 节点的配置文件名称是 `core`。

配置文件名称的后缀支持：`json`、`toml`、`yaml`、`properties`、`props`、`prop`，它们分别对应 JSON 文件、TOML 文件、YAML 文件和 Properties 文件，程序设置配置文件的时候如果不指定后缀，则按支持的后缀顺序在配置文件路径下进行搜索，找到第一个匹配的文件作

为最终的配置文件。文件路径和文件后缀按照广度优先的搜索顺序，即在同一路径下匹配完所有文件后缀再进入下一个文件路径。假设 Peer 节点没有设置配置文件路径，\$GOPATH 的路径是 /opt/gopath，则有两个目录下的文件如下所示：

```
vagrant@hyperledger-devenv:v0.2.2-58cde93: /opt/gopath/bin$ tree .
```

```
.
├── core.yaml
└── peer
```

```
vagrant@hyperledger-devenv:v0.2.2-58cde93: /opt/gopath/bin$ tree /opt/gopath/src/github.com/hyperledger/fabric/peer
```

```
.
├── core.json
└── peer
```

搜索路径顺序是：

```
./core.json
./core.toml
./core.yaml
./core.properties
./core.props
./core.prop
/opt/gopath/src/github.com/hyperledger/fabric/peer/core.json
/opt/gopath/src/github.com/hyperledger/fabric/peer/core.toml
/opt/gopath/src/github.com/hyperledger/fabric/peer/core.yaml
/opt/gopath/src/github.com/hyperledger/fabric/peer/core.properties
/opt/gopath/src/github.com/hyperledger/fabric/peer/core.props
/opt/gopath/src/github.com/hyperledger/fabric/peer/core.prop
```

所以最终获取到的配置文件是：./core.yaml，而不是 /opt/gopath/src/github.com/hyperledger/fabric/peer/core.json。

## 2.4 本章小结

本章零基础地介绍了如何快速体验超级账本搭建的区块链网络，我们先绕过了比较复杂的初始化配置，用官方提供的 fabric-samples 提供的配置和链码示例，展示了如何调用和查询链码，对 Hyperledger Fabric 实现的功能有一个初步的认识。这部分更为详细的初始化配置和网络部署等内容将在第 11 章会详细介绍。后面的章节我们会带你一起领略一下区块链的奥秘。

## 第二篇

## Part 2

# 核 心 篇

- 第3章 超级账本的系统架构
- 第4章 基于Gossip的P2P数据分发
- 第5章 分布式账本存储
- 第6章 集成共识机制的排序服务
- 第7章 实现数据隔离的多链及多通道
- 第8章 基于数字证书的成员管理服务
- 第9章 支持多种语言的智能合约



## 超级账本的系统架构

区块链的业务需求多种多样，一些要求在快速达成网络共识及快速确认区块后，才可以将区块加入区块链中。有一些可以接受相对缓慢的处理时间，以换取较低级别的信任。各行各业在扩展性、可信度、合法性、 workflow 复杂度以及安全性等方面的需求和用途都不尽相同。我们先来看一下在企业级区块链系统中常见的模块构成，如图 3-1 所示。

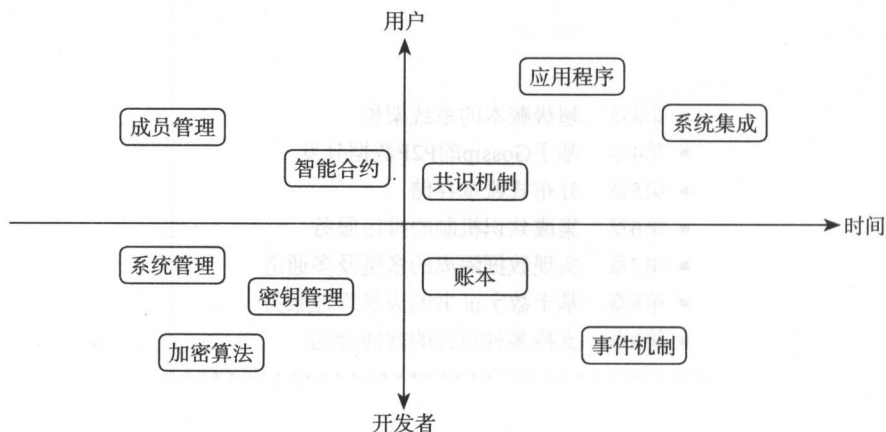


图 3-1 企业级区块链系统的常用功能

从图 3-1 中可以看到一些常用的功能模块有：应用程序、成员管理、智能合约、账本、共识机制、事件机制、系统管理等。纵轴代表用户或者开发者更关心的内容，越往上代表用户更关注，比如应用程序和钱包等，越靠下是开发者更关注的模块，比如事件机制。而横轴则是从时间的维度来看的，左边是一开始关注的功能，直到完成所有的功能。

Hyperledger Fabric 1.0 是一种通用的区块链技术，其设计目标是利用一些成熟的技术实现分布式账本技术（Distributed Ledger Technology, DLT）。超级账本采用模块化架构设计，复用通用的功能模块和接口。模块化的方法带来了可扩展性、灵活性等优势，会减少模块修改、升级带来的影响，能很好地利用微服务实现区块链应用系统的开发和部署。

Hyperledger Fabric 1.0 设计有几个特点：

1) **模块插件化**：很多的功能模块（如 CA 模块、共识算法、状态数据库存储、ESCC、VSCC、BCCSP 等）都是可插拔的，系统提供了通用的接口和默认的实现，这满足了大多数的业务需求。这些模块也可以根据需求进行扩展，集成到系统中。

2) **充分利用容器技术**：不仅节点使用容器作为运行环境，链码也默认运行在安全的容器中。应用程序或者外部系统不能直接操作链码，必须通过背书节点提供的接口转发给链码来执行。容器给链码运行提供的是安全沙箱环境，把链码的环境和背书节点的环境隔离开，链码存在安全问题也不会影响到背书节点。

3) **可扩展性**：Hyperledger Fabric 1.0 在 0.6 版本的基础上，对 Peer 节点的角色进行了拆分，有背书节点（Endorser）、排序服务节点（Orderer）、记账节点（Committer）等，不同角色的节点有不同的功能。节点可以加入到不同的通道（Channel）中，链码可以运行在不同的节点上，这样可以更好地提升并行执行的效率和吞吐量。

4) **安全性**：Hyperledger Fabric 1.0 提供的是授权访问的区块链网络，节点共同维护成员信息，MSP（Membership Service Provider）模块验证、授权了最终用户后才能使用区块链网络的功能。多链和多通道的设计容易实现数据隔离，也提供了应用程序和链码之间的安全通道，实现了隐私保护。

### 3.1 系统逻辑架构

图 3-2 所示为 Hyperledger Fabric 1.0 设计的系统逻辑架构图。

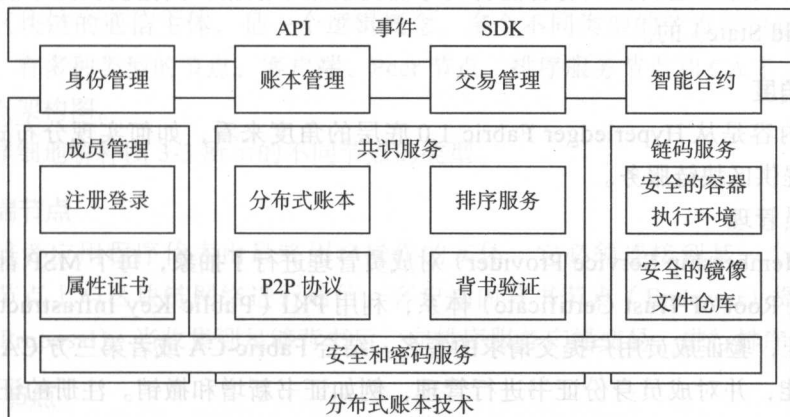


图 3-2 Hyperledger Fabric 1.0 的系统逻辑架构图

图 3-2 所示的系统逻辑架构图是从不同角度来划分的，上层从应用程序的角度，提供了标准的 gRPC 接口，在 API 的基础之上封装了不同语言的 SDK，包括 Golang、Node.js、Java、Python 等，开发人员可以利用 SDK 开发基于区块链的应用。区块链强一致性要求，各个节点之间达成共识需要较长的执行时间，也是采用异步通信的模式进行开发的，事件模块可以在触发区块事件或者链码事件的时候执行预先定义的回调函数。下面分别从应用程序和底层的角度分析应该关注的几个要素。

## 1. 应用程序角度

### (1) 身份管理

用户注册和登录系统后，获取到用户注册证书（ECert），其他所有的操作都需要与用户证书关联的私钥进行签名，消息接收方首先会进行签名验证，才进行后续的消息处理。网络节点同样会用到颁发的证书，比如系统启动和网络节点管理等都会对用户身份进行认证和授权。

### (2) 账本管理

授权的用户是可以查询账本数据（ledger）的，这可以通过多种方式查询，包括根据区块号查询区块、根据区块哈希查询区块、根据交易号查询区块、根据交易号查询交易，还可以根据通道名称获取查询到的区块链信息。

### (3) 交易管理

账本数据只能通过交易执行才能更新，应用程序通过交易管理提交交易提案（Proposal）并获取到交易背书（Endorsement）以后，再给排序服务节点提交交易，然后打包生成区块。SDK 提供接口，利用用户证书本地生成交易号，背书节点和记账节点都会校验是否存在重复交易。

### (4) 智能合约

实现“可编程的账本”（Programmable Ledger），通过链码执行提交的交易，实现基于区块链的智能合约业务逻辑。只有智能合约才能更新账本数据，其他模块是不能直接修改状态数据（World State）的。

## 2. 底层角度

下面的内容是从 Hyperledger Fabric 1.0 底层的角度来看，如何实现分布式账本技术，给应用程序提供区块链服务。

### (1) 成员管理

MSP（Membership Service Provider）对成员管理进行了抽象，每个 MSP 都会建立一套根信任证书（Root of Trust Certificate）体系，利用 PKI（Public Key Infrastructure）对成员身份进行认证，验证成员用户提交请求的签名。结合 Fabric-CA 或者第三方 CA 系统，提供成员注册功能，并对成员身份证书进行管理，例如证书新增和撤销。注册的证书分为注册证书（ECert）、交易证书（TCert）和 TLS 证书（TLS Cert），它们分别用于用户身份、交易

签名和 TLS 传输。

### (2) 共识服务

在分布式节点环境下,要实现同一个链上不同节点区块的一致性,同时要确保区块里的交易有效和有序。共识机制由3个阶段完成:客户端向背书节点提交提案进行签名背书,客户端将背书后的交易提交给排序服务节点进行交易排序,生成区块和排序服务,之后广播给记账节点验证交易后写入本地账本。网络节点的P2P协议采用的是基于Gossip的数据分发,以同一组织为传播范围来同步数据,提升网络传输的效率。

### (3) 链码服务

智能合约的实现依赖于安全的执行环境,确保安全的执行过程 and 用户数据的隔离。Hyperledger Fabric 采用 Docker 管理普通的链码,提供安全的沙箱环境和镜像文件仓库。其好处是容易支持多种语言的链码,扩展性很好。Docker 的方案也有自身的问题,比如对环境要求较高,占用资源较多,性能不高等,实现过程中也存在与 Kubernetes、Rancher 等平台的兼容性问题。

### (4) 安全和密码服务

安全问题是企业级区块链关心的问题,尤其在关注国家安全的项目中。其中底层的密码学支持尤其重要,Hyperledger Fabric 1.0 专门定义了一个 BCCSP (BlockChain Cryptographic Service Provider),使其实现密钥生成、哈希运算、签名验签、加密解密等基础功能。BCCSP 是一个抽象的接口,默认是软实现的国标算法,目前社区和较多的厂家都在实现国密的算法和 HSM (Hardware Security Module)。

Hyperledger Fabric 1.0 在架构上的设计具有很好的可扩展性,目前是众多可见的区块链技术中最为活跃的,值得区块链技术爱好者深入研究。

## 3.2 网络节点架构

节点是区块链的通信主体,是一个逻辑概念。多个不同类型的节点可以运行在同一物理服务器上。有多种类型的节点:客户端、Peer 节点、排序服务节点和 CA 节点。图 3-3 所示为网络节点架构图。

接下来详细地解释图 3-3 所示的不同节点的类型。

### 1. 客户端节点

客户端或者应用程序代表由最终用户操作的实体,它必须连接到某一个 Peer 节点或者排序服务节点上与区块链网络进行通信。客户端向背书节点 (Endorser) 提交交易提案 (Transaction Proposal),当收集到足够背书后,向排序服务广播交易,进行排序,生成区块。

### 2. Peer 节点

所有的 Peer 节点都是记账节点 (Committer),负责验证从排序服务节点区块里的交易,

维护状态数据和账本的副本。部分节点会执行交易并对结果进行签名背书，充当背书节点的角色。背书节点是动态的角色，是与具体链码绑定的。每个链码在实例化的时候都会设置背书策略，指定哪些节点对交易背书后才是有效的。也只有在应用程序向它发起交易背书请求的时候才是背书节点，其他时候就是普通的记账节点，只负责验证交易并记账。

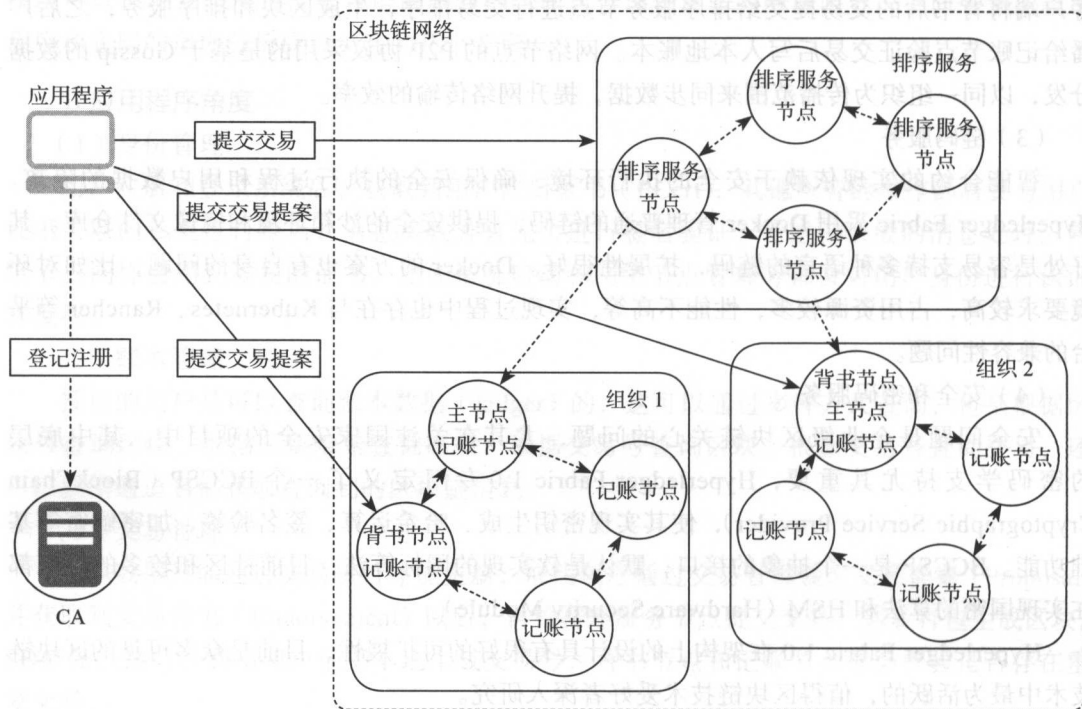


图 3-3 网络节点架构图

图 3-2 所示的 Peer 节点还有一种角色是主节点（Leader Peer），代表的是和排序服务节点通信的节点，负责从排序服务节点处获取最新的区块并在组织内部同步。可以强制设置为主节点，也可以动态选举产生。

在图 3-2 中还可以看到，有的节点同时是背书节点和记账节点，也可以同时是背书节点、主节点和记账节点，也可以只是记账节点。在后面的章节中，有的地方会用记账节点代表普通的 Peer 节点。

### 3. 排序服务节点

排序服务节点（Ordering Service Node 或者 Orderer）接收包含背书签名的交易，对未打包的交易进行排序生成区块，广播给 Peer 节点。排序服务提供的是原子广播（Atomic Broadcast），保证同一个链上的节点接收到相同的消息，并且有相同的逻辑顺序。

排序服务的多通道（MultiChannel）实现了多链的数据隔离，保证只有同一个链的 Peer 节点才能访问链上的数据，保护用户数据的隐私。

排序服务可以采用集中式服务，也可以采用分布式协议。可以实现不同级别的容错处理，目前正式发布的版本只支持 Apache Kafka 集群，提供交易排序的功能，只实现 CFT (Crash Fault Tolerance，崩溃故障容错)，不支持 BFT (Byzantine Fault Tolerance，拜占庭容错)。

4. CA 节点

CA 节点是 Hyperledger Fabric 1.0 的证书颁发机构 (Certificate Authority)，由服务器和客户端组件组成。CA 节点接收客户端的注册申请，返回注册密码用于用户登录，以便获取身份证书。在区块链网络上所有的操作都会验证用户的身份。CA 节点是可选的，可以用其他成熟的第三方 CA 颁发证书。

3.3 典型交易流程

图 3-4 所示为 Hyperledger Fabric 1.0 典型的交易流程图。

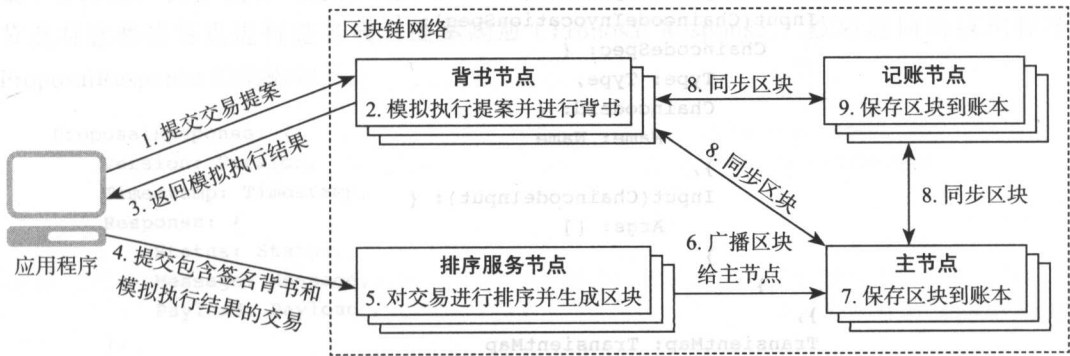


图 3-4 交易流程总图

从上一节的网络节点架构中，我们已经了解到基于 Hyperledger Fabric 1.0 的区块链应用中涉及几个节点角色：应用程序、背书节点、排序服务节点和主节点。在图 3-4 中，假定各节点已经提前颁发好证书，且已正常启动，并加入已经创建好的通道。后面的步骤介绍在已经实例化了的链码通道上从发起一个调用交易到最终记账的全过程。

3.3.1 创建交易提案并发送给背书节点

使用应用程序构造交易提案，SignedProposal 的结构如下所示：

```
SignedProposal: {
  ProposalBytes(Proposal): {
    Header: {
      ChannelHeader: {
        Type: "HeaderType_ENDORSER_TRANSACTION",
```



```

    TxId: TxId,
    Timestamp: Timestamp,
    ChannelId: ChannelId,
    Extension(ChaincodeHeaderExtension): {
        PayloadVisibility: PayloadVisibility,
        ChaincodeId: {
            Path: Path,
            Name: Name,
            Version: Version
        }
    },
    Epoch: Epoch
},
SignatureHeader: {
    Creator: Creator,
    Nonce: Nonce
}
},
Payload: {
    ChaincodeProposalPayload: {
        Input(ChaincodeInvocationSpec): {
            ChaincodeSpec: {
                Type: Type,
                ChaincodeId: {
                    Name: Name
                },
                Input(ChaincodeInput): {
                    Args: []
                }
            },
            TransientMap: TransientMap
        }
    }
},
Signature: Signature
}

```

我们来看看上面的结构，SignedProposal 是封装了 Proposal 的结构，添加了调用者的签名信息。背书节点会根据签名信息验证其是否是一个有效的消息。Proposal 由两个部分组成：消息头和消息结构。消息结构详细的解释参考后面的章节。这里简单讲一下消息头。

消息头 (Header) 也包含两项内容。

1) 通道头 (ChannelHeader)：通道头包含了与通道和链码调用相关的信息，比如在哪个通道上调用哪个版本的链码。TxId 是应用程序本地生成的交易号，跟调用者的身份证书相关，可以避免交易号的冲突，背书节点和记账节点都会校验是否存在重复交易。

2) 签名头 (SignatureHeader)：签名头包含了调用者的身份证书和一个随机数，用于消息的有效性校验。

应用程序构造好交易提案请求后,选择背书节点执行并进行背书签名。背书节点是链码背书策略里指定的节点。有一些背书节点是离线的,其他的背书节点可以拒绝对交易进行背书,也可以不背书。应用程序可以尝试使用其他可用的背书节点来满足策略。应用程序以何种顺序给背书节点发送背书请求是没有关系的,正常情况下背书节点执行后的结果是一致的,只有背书节点对结果的签名不一样。

### 3.3.2 背书节点模拟交易并生成背书签名

背书节点在收到交易提案后会进行一些验证,包括:

- ☐ 交易提案的格式是否正确;
- ☐ 交易是否提交过(重复攻击保护);
- ☐ 交易签名有效(通过 MSP);
- ☐ 交易提案的提交者在当前通道上是否已授权有写权限。

验证通过后,背书节点会根据当前账本数据模拟执行链码中的业务逻辑并生成读写集(RwSet),其中包含响应值、读写集等。在模拟执行时账本数据不会更新。而后背书节点对这些读写集进行签名成为提案响应(Proposal Response),然后返回给应用程序。ProposalResponse 的结构如下:

```
ProposalResponse: {
    Version: Version,
    Timestamp: Timestamp,
    Response: {
        Status: Status,
        Message: Message,
        Payload: Payload
    },
    Payload(PayloadResponsePayload): {
        ProposalHash: ProposalHash,
        Extension(ChaincodeAction): {
            Results(TxRwSet): {
                NsRwSets(NsRwSet): [
                    Namespace: Namespace,
                    KvRwSet: {
                        Reads(KVRead): [
                            Key: Key,
                            Version: {
                                BlockNum: BlockNum,
                                TxNum: TxNum
                            }
                        ]
                    },
                    RangeQueriesInfo(RangeQueryInfo): [
                        StartKey: StartKey,
                        EndKey: EndKey,
                        ItrExhausted: ItrExhausted,
                        ReadsInfo: ReadsInfo
                    ]
                ]
            }
        }
    }
}
```

```

    ],
    Writes(KVWrite): [
        Key: Key,
        IsDelete: IsDelete,
        Value: Value
    ]
}

},
Events(ChaincodeEvent): {
    ChaincodeId: ChaincodeId,
    TxId: TxId,
    EventName: EventName,
    Payload: Payload
}
Response: {
    Status: Status,
    Message: Message,
    Payload: Payload
},
ChaincodeId: ChaincodeId
}
},
Endorsement: {
    Endorser: Endorser,
    Signature: Signature
}
}
}

```

返回的 `ProposalResponse` 中包含了读写集、背书节点签名以及通道名称等信息，更多字段的详细解释参考 3.4 节。

背书节点接收消息后执行的详细过程请参考第 9 章的相关内容。

### 3.3.3 收集交易的背书

应用程序收到 `ProposalResponse` 后会对背书节点签名进行验证，所有节点接收到任何消息后都是需要先验证消息合法性的。如果链码只进行账本查询，应用程序会检查查询响应，但不会将交易提交给排序服务节点。如果链码对账本进行 `Invoke` 操作，则须提交交易给排序服务进行账本更新，应用程序会在提交交易前判断背书策略是否满足。如果应用程序没有收集到足够的背书就提交交易了，记账节点在提交验证阶段会发现交易不能满足背书策略，标记为无效交易。

如何选择背书节点呢？目前 `fabric-sdk-go` 默认的实现是把配置文件选项 `channels.mychannel.peers`（其中的 `mychannel` 需要替换成实际的通道名称）里的节点全部添加为背书节点，需要等待所有背书节点的背书签名。应用程序等待每个背书节点执行的超时时间是通过配置文件选项 `client.peer.timeout.connection` 设置的，配置文件的示例给出的是 3 秒，

根据实际情况调整，如果没有设置就是5秒的默认值。

### 3.3.4 构造交易请求并发送给排序服务节点

应用程序接收到所有的背书节点签名后，根据背书签名调用 SDK 生成交易，广播给排序服务节点。生成交易的过程比较简单，确认所有的背书节点的执行结果完全一致，再将交易提案、提案响应和背书签名打包生成交易。交易的结构如下：

```
Envelope: {
  Payload: {
    Header: {
      ChannelHeader: {
        Type: "HeaderType_ENDORSER_TRANSACTION",
        TxId: TxId,
        Timestamp: Timestamp,
        ChannelId: ChannelId,
        Extension(ChaincodeHeaderExtension): {
          PayloadVisibility: PayloadVisibility,
          ChaincodeId: {
            Path: Path,
            Name: Name,
            Version: Version
          }
        },
        Epoch: Epoch
      },
      SignatureHeader: {
        Creator: Creator,
        Nonce: Nonce
      }
    },
    Data(Transaction): {
      TransactionAction: [
        Header(SignatureHeader): {
          Creator: Creator,
          Nonce: Nonce
        },
        Payload(ChaincodeActionPayload): {
          ChaincodeProposalPayload: {
            Input(ChaincodeInvocationSpec): {
              ChaincodeSpec: {
                Type: Type,
                ChaincodeId: {
                  Name: Name
                },
                Input(ChaincodeInput): {
                  Args: []
                }
              }
            }
          }
        }
      ]
    }
  }
}
```

```

    },
    TransientMap: nil
  },
  Action(ChaincodeEndorsedAction): {
    Payload(ProposalResponsePayload): {
      ProposalHash: ProposalHash,
      Extension(ChaincodeAction): {
        Results(TxRwSet): {
          NsRwSets(NsRwSet): [
            Namespace: Namespace,
            KvRwSet: {
              Reads(KVRead): [
                Key: Key,
                Version: {
                  BlockNum: BlockNum,
                  TxNum: TxNum
                }
              ],
              RangeQueriesInfo(RangeQueryInfo): [
                StartKey: StartKey,
                EndKey: EndKey,
                ItrExhausted: ItrExhausted,
                ReadsInfo: ReadsInfo
              ],
              Writes(KVWrite): [
                Key: Key,
                IsDelete: IsDelete,
                Value: Value
              ]
            }
          ]
        }
      },
      Events(ChaincodeEvent): {
        ChaincodeId: ChaincodeId,
        TxId: TxId,
        EventName: EventName,
        Payload: Payload
      }
    },
    Response: {
      Status: Status,
      Message: Message,
      Payload: Payload
    },
    ChaincodeId: ChaincodeId
  },
  Endorsement: [
    Endorser: Endorser,
    Signature: Signature
  ]
}

```

```

    }
  }
},
Signature: Signature
}

```

我们来看交易信封的几个对应关系：

- ❑ `Envelope.Payload.Header` 同交易提案 `SignedProposal.Proposal.Header`;
- ❑ `Envelope.Payload.Data.TransactionAction.Header` 是交易提案的提交者的身份信息，同 `SignedProposal.Proposal.Header.SignatureHeader` 和 `Envelope.Payload.Header.SignatureHeader` 是冗余的；
- ❑ `Envelope.Payload.Data.TransactionAction.Payload.ChaincodeProposalPayload` 同交易提案的 `SignedProposal.Proposal.Payload.ChaincodeProposalPayload`，唯一不同的是，`TransientMap` 强制设置为 `nil`，目的是避免在区块中出现一些敏感信息；
- ❑ `Envelope.Payload.Data.TransactionAction.Payload.Action.Payload` 结构，其实和 `ProposalResponse.Payload` 结构完全一样；
- ❑ `Envelope.Payload.Data.TransactionAction.Payload.Action.Endorsement` 变成了数组，代表多个背书节点的背书签名。

整个信封 `Envelope` 的 `Signature` 是交易提交者对整个 `Envelope.Payload` 的签名。应用程序可以把生成的交易信封内容发送给任意选择的几个排序服务节点。

### 3.3.5 排序服务节点以对交易进行排序并生成区块

排序服务不读取交易的内容，如果在生成交易信封内容的时候伪造了交易模拟执行的结果，排序服务节点也不会发现，但会在最终的交易验证阶段校验出来并标记为无效交易。排序服务要做得很简单，先是接收网络中所有通道发出的交易信息，读取交易信封的 `Envelope.Payload.Header.ChannelHeader.ChannelId` 以获取通道名称，按各个通道上交易的接收时间顺序对交易信息进行排序，生成区块，详细的流程请参考第6章。

### 3.3.6 排序服务节点以广播给组织的主节点

排序服务节点生成区块以后会广播给通道上不同组织的主节点，详细的流程请参考第4章。

### 3.3.7 记账节点验证区块内容并写入区块

背书节点是动态角色，只要参与交易的背书就是背书节点，哪些交易选择哪些节点作为背书节点是由应用程序选择的，这需要满足背书策略才能生效。所有的背书节点都属于



记账节点。所有的 Peer 节点都是记账节点，记录的是节点已加入通道的账本数据。记账节点接收到的是排序服务节点生成的区块，验证区块交易的有效性，提交到本地账本后再产生一个生成区块的事件，监听区块事件的应用程序可以进行后续的处理。如果接收到的区块是配置区块，则会更新缓存的配置信息。记账节点的处理流程如图 3-5 所示。

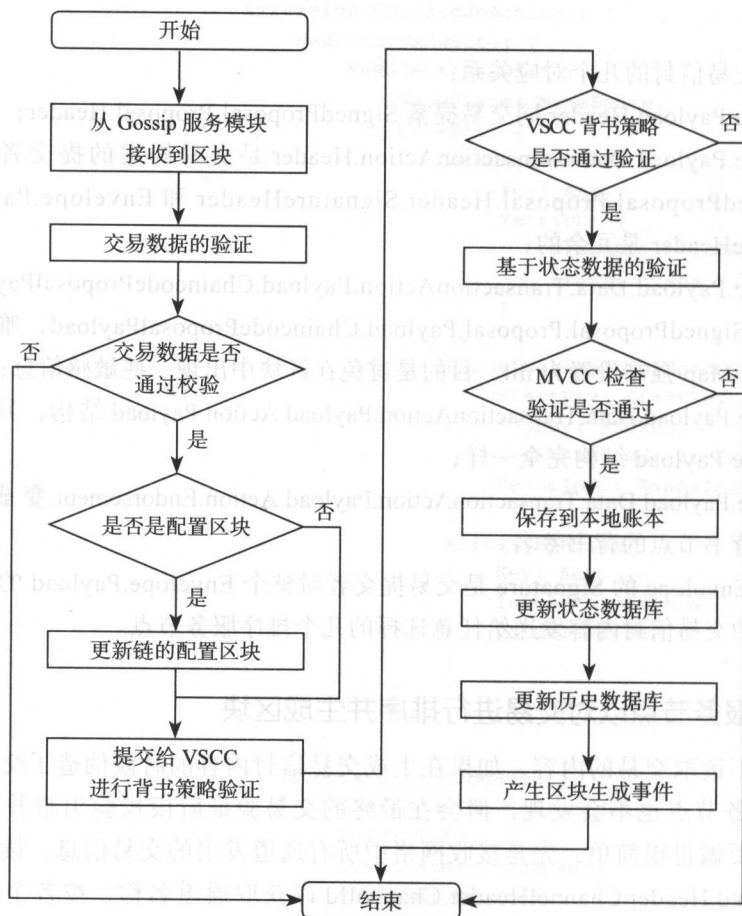


图 3-5 记账节点的流程图

### 1. 交易数据的验证

区块数据的验证是以交易验证为单位的，每次对区块进行验证时都会生成一个交易号的位图 TxValidationFlags，它记录每个交易号的交易验证状态，只有状态为 TxValidationCode\_VALID 才是有效的。位图也会写入到区块的元数据 BlockMetadataIndex\_TRANSACTIONS\_FILTER 中。交易验证的时候会检查以下内容：

- ❑ 是否为合法的交易：交易格式是否正确，是否有合法的签名，交易内容是否被篡改；
- ❑ 记账节点是否加入了这个通道。

基本的验证通过以后会提交给 VSCC 进行背书策略的验证。

## 2. 记账节点与 VSCC

链码的交易是隔离的，每个交易的模拟执行结果读写集 TxRwSet 都包含了交易所属的链码。为了避免错误地更新链码交易数据，在交易提交给系统链码 VSCC 验证交易内容之前，还会对链码进行校验。下面这些交易都是非法的：

- ☐ 链码的名称或者版本为空；
- ☐ 交易消息头里的链码名称 Envelope.Payload.Header.ChannelHeader.Extension.ChaincodeId.Name 和交易数据里的链码名称 Envelope.Payload.Data.TransactionAction.Payload.ChaincodeProposalPayload.Input.ChaincodeSpec.ChaincodeId.Name 不一致；
- ☐ 链码更新当前链码数据时，生成读写集的链码版本不是 LSCC 记录的最新版本；
- ☐ 应用程序链码更新了 LSCC（生命周期管理系统链码）的数据；
- ☐ 应用程序链码更新了不可被外部调用的系统链码的数据；
- ☐ 应用程序链码更新了不可被其他链码调用的系统链码的数据；
- ☐ 调用了不可被外部调用的系统链码。

更多系统链码的介绍，请参考 9.3 节。

## 3. 基于状态数据的验证和 MVCC 检查

交易通过 VSCC 检查以后，就进入记账流程。kvledger 还会对读写集 TxRwSet 进行 MVCC（Multi-Version Concurrency Control）检查。

kvledger 实现的是基于键值对（key-value）的状态数据模型。对状态数据的键有 3 种操作：

- ☐ 读状态数据；
- ☐ 写状态数据；
- ☐ 删除状态数据。

对状态数据的读操作有两种形式：

- ☐ 基于单一键的读取；
- ☐ 基于键范围的读取。

MVCC 检查只对读数据进行校验，基本逻辑是对模拟执行时状态数据的版本和提交交易时状态数据的版本进行比较。如果数据版本发生变化或者某个键的范围数据发生变化，就说明这段时间之内有别的交易改变了状态数据，当前交易基于原有状态的处理就是有问题。由于交易提交是并行的，所以在交易未打包生成区块之前，并不能确定最终的执行顺序。如果交易执行的顺序存在依赖，在 MVCC 检查的时候就会出现依赖的状态发生了变化，实际上是数据出现了冲突。图 3-6 所示为基于状态的数据验证的流程图。

写集合本身包含了写和删除的数据，有一个状态位标识是否删除数据。为了提升效率，状态数据库的提交是批处理的，整个区块交易的状态数据同时提交，这也保证了整个区块的状态数据要么都提交成功，要么都提交失败。这时只会出现记录的账本数据和状态数据

库不一致，不会出现区块的状态数据不一致的情况。当账本数据和状态数据库不一致时，可以通过状态数据库的检查点来标记。

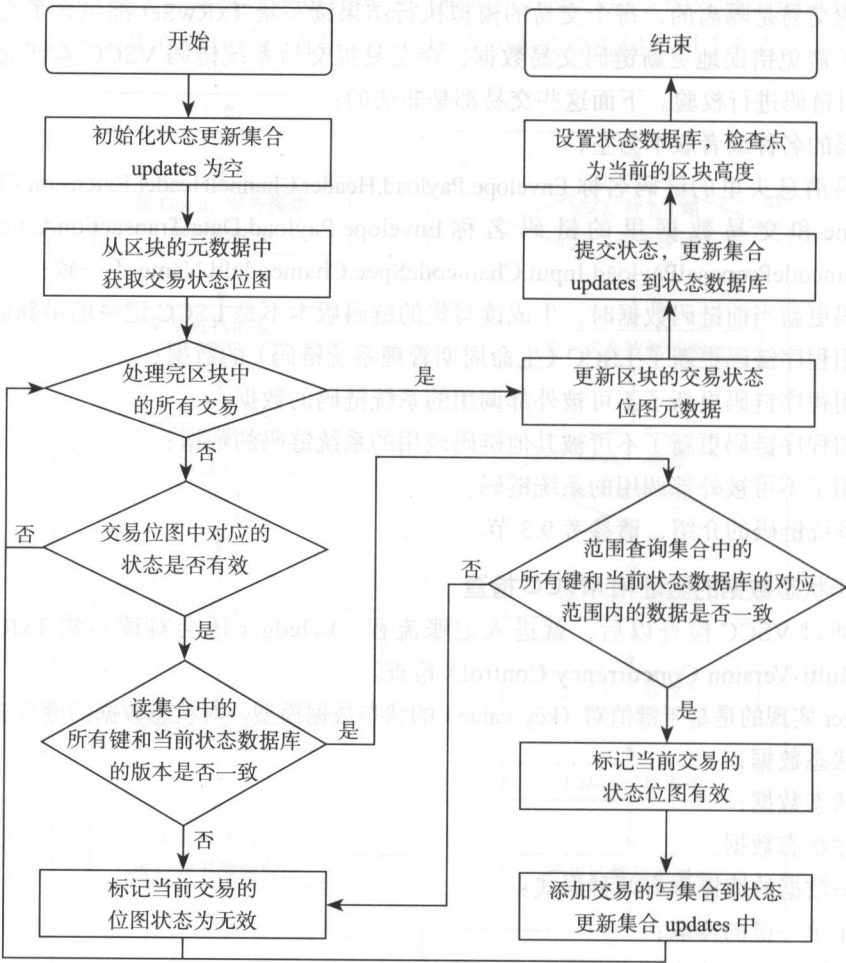


图 3-6 基于状态的数据验证的流程图

#### 4. 无效交易的处理

伪造的交易会导致无效交易，正常的交易也可能出现无效交易。MVCC 检查的是背书节点在模拟执行的时候，环境是否和记账节点提交交易时的环境一致，这里的环境是指状态数据库里数据的三元组（key、value、version）是否完全一致。如果正常提交的交易在这个过程中涉及的数据发生了变化，那么也会出现检查失败从而导致无效交易。在这种情况下，需要在上层的应用程序有一些补偿措施，比如调整交易打包的配置，重新提交失败的交易等。

在目前版本的实现中，无效交易也会保留在区块中，可以通过区块记录的元数据确定

哪些是无效交易。无效交易的读写集不会提交到状态数据库中，不会导致状态数据库发生变化，只是会占用区块的大小，占用记账节点的硬盘空间。后续的版本会实现账本的精简，过滤掉无效交易。

### 3.3.8 在组织内部同步最新的区块

主节点在组织内部同步区块的过程详见第4章的相关内容。

## 3.4 消息协议结构

### 3.4.1 信封消息结构

信封消息是认证内容中最基本的单元。它由一个消息负载（Payload）和一个签名（Signature）组成。

// 信封包含一个带有签名的负载，以便认证该消息

```
message Envelope {
```

```
    // 编组的负载
```

```
    bytes payload = 1;
```

```
    // 负载头中指定创建者签名
```

```
    bytes signature = 2;
```

```
}
```

```
    // 负载是消息内容（允许签名）
```

```
message Payload {
```

```
    // 负载头部，提供身份验证并防止重放
```

```
    Header header = 1;
```

```
    // 数据，其编码由头的类型定义
```

```
    bytes data = 2;
```

```
}
```

负载包含：

1) 消息头部。头部带有类型，描述负载的性质以及如何解组数据字段。此外，头部还包含创建者的信息和随机数，以及用来标识时间逻辑窗口的时期信息。只有在两个条件都成立的情况下，Peer 节点才能接受一个信封。

① 消息中指定的时期信息是当前窗口期；

② 该负载在该周期内只看到一次（即没有重放）。

2) 数据字段的类型由头部指定。头部消息的组织方式如下所示：

```
message Header {
```

```
    bytes channel_header = 1;
```

```
    bytes signature_header = 2;
```

```

}
// 通道头是一个通用的预防重放和身份标识的消息，它包含在一个被签名的负载之中
message ChannelHeader {
    int32 type = 1; // 头类型 0-10000 由 HeaderType 保留和定义

    // 版本指示消息协议版本
    int32 version = 2;

    // 时间戳是发件人创建消息的本地时间
    google.protobuf.Timestamp timestamp = 3;

    // 该消息绑定通道的标识符
    string channel_id = 4;

    // 端到端使用唯一的标识符
    // - 由较高层设置，如最终用户或 SDK
    // - 传递给背书节点（将检查唯一性）
    // - 当消息正确传递时，它将被记账节点检索（也会检查唯一性）
    // - 存储于账本中
    string tx_id = 5;

    // 时期信息基于区块高度而定义，此字段标识时间的逻辑窗口
    // 只有在两个条件都成立的情况下，对方才接受提案响应
    // 1. 消息中指定的时期信息是当前时期
    // 2. 该消息在该时期内只看到一次（即没有重放）
    uint64 epoch = 6;

    // 根据消息头类型附加的扩展
    bytes extension = 7;
}

enum HeaderType {
    MESSAGE = 0; // 非透明消息
    CONFIG = 1; // 通道配置
    CONFIG_UPDATE = 2; // 通道配置更新
    ENDORSER_TRANSACTION = 3; // SDK 提交背书
    ORDERER_TRANSACTION = 4; // 排序管理内部使用
    DELIVER_SEEK_INFO = 5; // 指示 Deliver API 查找信息
    CHAINCODE_PACKAGE = 6; // 链码打包安装
}

message SignatureHeader {
    // 消息的创建者，链的证书
    bytes creator = 1;

    // 只能使用一次的任意数字，可用于检测重放攻击
    bytes nonce = 2;
}

```

信封消息结构对于验证负载的签名是必要的。否则，对于大载荷消息，就必须连接所有的载荷再进行签名验证，这往往成本很高。

经过排序后，批量的信封消息交付给记账节点进行验证，通过验证后的数据被记录到账本之中。

### 3.4.2 配置管理结构

区块链有与之相关的配置，配置设置在创世区块之中，但可能在后续被修改。该配置信息在类型为 CONFIGURATION\_TRANSACTION 的信封消息中编码。配置信息本身就是区块的一个单独交易。配置信息交易没有任何依赖，所以每个配置信息交易必须包含对于链的全量数据，而不是增量数据。使用全量数据更容易引导新的 peer 或排序节点，也便于未来进行裁剪工作。

CONFIGURATION\_TRANSACTION 类型的信封消息具有 ConfigurationEnvelope 类型的负载数据。它定义为：

```
message ConfigurationEnvelope {
    repeated SignedConfigurationItem Items = 3;
}
```

配置信息的信封消息有与之关联的序列号和链 ID。每次修改配置序列号必须递增，这可以作为防止重放攻击的一个简单机制。配置信息的信封中会嵌入一系列的 SignedConfigurationItems，定义如下：

```
message SignedConfigurationItem {
    bytes ConfigurationItem = 1;
    repeated Envelope Signatures = 2;
}
```

因为 SignedConfigurationItem 必须支持多个签名，所以它包含一组重复的信封消息。这些消息中每个都有一个类型为 CONFIGURATION\_ITEM 的头部。负载的数据部分在 ConfigurationItem 中保存，定义为：

```
message ConfigurationItem {
    enum ConfigurationType {
        Policy = 0;
        Chain = 1;
        Orderer = 2;
        Fabric = 3;
    }
    bytes ChainID = 1;
    uint64 LastModified = 2;
    ConfigurationType Type = 3;
    string ModificationPolicy = 4;
    string Key = 5;
    bytes Value = 6;
}
```



Type 提供了配置项的范围和编码信息。LastModified 字段设置为上一次配置项被修改时配置信息信封中的序列号。ModificationPolicy 指向一个已经命名的策略，用来对将来的签名进行评估，以确定修改是否被授权。Key 和 Value 字段分别用作引用配置项及其内容。

修改配置包含以下内容：

- ☐ 检索现有配置；
- ☐ 递增配置信息信封消息中的序列号；
- ☐ 修改所需的配置项，将每个配置项的 LastModified 字段设置为递增后的序列号；
- ☐ 更新 SignedConfigurationItem 中的签名信息；
- ☐ 将签名后的信封信息提交给排序节点。

配置管理员将验证：

- ☐ 所有配置项和信封都指向正确的链；
- ☐ 添加或修改了哪些配置项；
- ☐ 有没有现有的配置项被忽略（即提交的数据是全集）；
- ☐ 所有配置更改的 LastModification 都等于信封消息中的序列号；
- ☐ 所有配置更改均符合相应的修改策略。

任何有权更改配置项的角色都可以构建新的配置信息交易。修改配置项将更新序列号并产生新的创世区块，这将引导新加入网络的各种节点。

### 3.4.3 背书流程结构

当 Envelope.data 中携带与链码相关的消息时，使用 ENDORSER\_TRANSACTION 类型。

获得批准的 ENDORSER\_TRANSACTION 负载流程如下。

首先，客户端向所有相关的背书节点发送提案消息（提案基本上是要进行一些影响账本的动作）。

然后，每个背书节点向客户端发送一个提案响应消息。提案响应包含背书结果的成功/错误码、应答负载和签名。应答负载之中包含提案的哈希值信息，用此信息可以将提案和针对该提案的应答安全地连接起来。

最后，客户端将背书结果写入交易中，签名并发送到排序服务。

在接下来的章节中，我们将详细介绍消息及其流程。

#### 1. 交易提案结构

一个提案消息包含头部（包含描述它的一些元数据，例如类型、调用者的身份、时间、链的 ID、加密的随机数）和不透明的负载：

```
message SignedProposal {
    // 提案
    bytes proposal_bytes = 1;
```

```
// 对提案进行签名, 该签名将和头部的创建者标识进行验证
bytes signature = 2;
}
message Proposal {
    // 提案头部
    bytes header = 1;
    // 提案负载, 具体结构由头部的类型决定
    bytes payload = 2;
    // 提案的可选扩展。对于 CHAINCODE 类型的消息, 其内容可能是 ChaincodeAction 消息
    bytes extension = 3;
}
```

一个提案发送给背书节点进行背书。该提案包含:

- 头部, 可以解组为头部信息;
- 负载, 由头部的类型决定;
- 扩展, 由头部的类型决定。

和信封消息结构类似, 这种 SignedProposal 消息结构也是重要的。否则, 对于大载荷消息, 我们必须连接所有的载荷再进行签名验证, 这往往成本很高。

当背书节点收到签名后的提案消息后, 它将验证消息中的签名。验证签名需要以下步骤。

1) 预验证用户生成签名证书的有效性。一旦 SignedProposal.proposal\_bytes 和 Proposal.header 都解组成功, 就可以认为证书基本是可用的。虽然这种在证书验证前的解组操作可能并不太理想, 但是在这个阶段可以过滤掉证书过期的情况。

2) 验证证书是否可信 (证书是否由受信任的 CA 签名), 并允许交易 (能否通过 ACL 检查)。

3) 验证 SignedProposal.proposal\_bytes 上的签名是否有效。

4) 检测重放攻击。

以下是当 ChainHeader 的类型为 ENDORSER\_TRANSACTION 时的消息:

```
message ChaincodeHeaderExtension {
    // 控制提案的负载在最终交易和账本中的可见程度
    bytes payload_visibility = 1;
    // 要定位的链代码 ID
    ChaincodeID chaincode_id = 2;
}
```

ChaincodeHeaderExtension 消息用于指定要调用的链码以及应在账本中呈现的内容。理想情况下, payload\_visibility 是可配置的, 支持至少 3 种主要可见性模式:

- 负载所有字节都可见;
- 只有负载的哈希值可见;
- 任何东西都不可见。

注意，可见性功能可能也会由 ESCC 设置，此时本字段将会被覆盖。另外本字段也将影响 ProposalResponsePayload.proposalHash 的内容。

```
message ChaincodeProposalPayload {
    // 包含调用链码的参数，
    bytes input = 1;
    // 用于实现某些应用程序级的加密数据
    map<string, bytes> TransientMap = 2;
}
```

ChaincodeProposalPayload 消息包含调用链码的参数。TransientMap 字段的内容应始终从信封消息中省略掉，并不记录到账本之中。

```
message ChaincodeAction {
    // 调用链码产生的读 / 写集
    bytes results = 1;
    // 调用链码产生的事件
    bytes events = 2;
    // 调用链码的结果
    Response response = 3;
    // 含链 ID、背书节点在模拟执行提案时设置
    // 账本节点将验证版本号是否与链码最新版本匹配，含有链 ID 信息将支持单个交易打开多个链码
    ChaincodeID chaincode_id = 4;
}
```

ChaincodeAction 消息包含执行链码所产生的动作和事件。results 字段包含读取集合，events 字段包含由链码执行生成的事件。

## 2. 提案响应结构

提案响应消息从背书节点返回给提案客户端。背书节点使用该消息表达对于交易提案的处理结果。应答结果可能是成功也可能是失败，另外还会包含动作描述和背书节点的签名。如果足够数量的背书节点同意相同的动作并进行签名，则可以生成负载消息，并发送给排序节点。

```
message ProposalResponse {
    // 消息协议版本
    int32 version = 1;
    // 消息创建时间，由消息发送者定义
    google.protobuf.Timestamp timestamp = 2;
    // 某个动作的背书是否成功
    Response response = 4;
    // 负载，ProposalResponsePayload 字节序列
    bytes payload = 5;
    // 提案的具体背书内容，基本上就是背书节点的签名
    Endorsement endorsement = 6;
}

message ProposalResponsePayload {
    // 触发此应答交易提案的哈希值
```

```

bytes proposal_hash = 1;
// 扩展内容, 应该解组为特定类型的消息
bytes extension = 2;
}
message Endorsement {
    // 背书节点身份 (例如, 证书信息)
    bytes endorser = 1;
    // 签名, 对提案应答负载和背书节点证书这两个内容进行签名
    bytes signature = 2;
}

```

ProposalResponsePayload 消息是提案响应的负载部分。这个消息是客户端请求和背书节点动作之间的“桥梁”。对于链码来说, 它包含一个表示提议的哈希值 `proposal_hash`, 以及表示链码状态变化和事件 `extension` 字段。

`proposal_hash` 字段将交易提案和提案响应两者对应起来, 即为了实现异步系统的记账功能也为了追责和抗抵赖的安全诉求。哈希值通常会覆盖整个提案消息的所有字节中。但是, 这样实现就意味着只有获得完整的提案消息才能验证哈希值的正确性。

出于保密原因, 使用链码不太可能将提案的负载直接存储在账本中。例如, 类型为 `ENDORSER_TRANSACTION` 的消息, 需要将提案的头部和负载分开进行处理: 头部总是进行完整散列的, 而负载则可能进行完整散列或对哈希值再进行散列, 或者根本不进行散列。

### 3. 背书交易结构

客户端获得足够的背书后, 可以将这些背书组合成一个交易信息。这个交易信息可以设置为负载信息的数据字段。以下是在这种情况下要使用的具体消息:

```

message Transaction {
    // 负载是一个 TransactionAction 数组, 每个交易需要一个数组来适应多个动作
    repeated TransactionAction actions = 1;
}

message TransactionAction {
    // 提案头部
    bytes header = 1;
    // 负载由头部类型决定, 它是 ChaincodeActionPayload 字节序列
    bytes payload = 2;
}

```

TransactionAction 消息将提案绑定到其动作。它的头部是 SignatureHeader 消息, 它的负载是 ChaincodeActionPayload 消息。

```

message ChaincodeActionPayload {
    // ChaincodeProposalPayload 消息的字节序列, 内容来自链码原始调用的参数
    bytes chaincode_proposal_payload = 1;
    // 应用于账本的动作列表
    ChaincodeEndorsedAction action = 2;
}

```

ChaincodeActionPayload 消息携带 chaincodeProposalPayload 和已经通过背书的动作以应用于账本。主要的可见性模式是“full”（整个 ChaincodeProposalPayload 消息包含在这里）、“hash”（仅包含 ChaincodeProposalPayload 消息的哈希值）或“nothing”。该字段将用于检查 ProposalResponsePayload.proposalHash 的一致性。此外，action 字段包含应用于账本的动作列表。

```
message ChaincodeEndorsedAction {
    // 由背书节点签名的 ProposalResponsePayload 消息字节序列
    bytes proposal_response_payload = 1;
    // 提案背书，基本上是背书节点的签名
    repeated Endorsement endorsements = 2;
}
```

ChaincodeEndorsedAction 消息承载有关具体提案的背书信息。proposalResponsePayload 是由背书节点签名的，对于 ENDORSER\_TRANSACTION 类型，ProposalResponsePayload 的 extension 字段会带有一个 ChaincodeAction。此外，endorsements 字段包含提案已经收到的背书信息。

## 3.5 策略管理和访问控制

在 Hyperledger Fabric 1.0 中，较多的地方都使用策略进行管理，它是一种权限管理的方法，包括交易背书策略、链码的实例化策略、通道管理策略等。

### 3.5.1 策略定义及其类型

策略定义了一些规则，验证签名数据是否符合定义的条件，结果为 TRUE 或者 FALSE。策略的定义如下：

```
type Policy struct {
    Type int32 // 策略的类型
    Value []byte // 策略的内容
}
```

策略的类型有两种。

1) SignaturePolicy：在基于验证签名策略的基础上，支持条件 AND、OR、NOutOf 的任意组合，其中的 NOutOf 指的是满足  $m$  个条件中的  $n$  个就表示满足策略 ( $m \geq n$ )。比如 OR(Org1.Admin, NOutOf(2, Org2.Member)) 表示 Org1 的管理员或者两个 Org2 的成员签名都满足策略。

2) ImplicitMetaPolicy：隐含的元策略，是在 SignaturePolicy 之上的策略，支持大多数的组织管理员这种策略，只适用于通道管理策略。

SignaturePolicy 实际只有两种类型，SignedBy 和 NOutOf，其他的，比如 AND 和 OR

都会转换成 NOutOf 类型。其定义如下：

```
type SignaturePolicy struct {
    // 支持的类型有：
    // *SignaturePolicy_SignedBy, 验证单个签名是否正确
    // *SignaturePolicy_NOutOf_, 验证是否有 n 个签名都正确
    Type isSignaturePolicy_Type `protobuf_oneof:"Type"`
}
```

ImplicitMetaPolicy 是递归策略的定义方法，名称中的 Implicit 说明规则是由子策略生成的，Meta 说明策略依赖其他策略的验证结果。

```
type ImplicitMetaPolicy struct {
    SubPolicy string // 子策略的名称
    Rule ImplicitMetaPolicy_Rule // 策略的规则
}
```

策略的规则支持 3 种形式：

- ❑ ImplicitMetaPolicy\_ANY：任意一个子规则成立就满足策略；
- ❑ ImplicitMetaPolicy\_ALL：全部子规则都成立才满足策略；
- ❑ ImplicitMetaPolicy\_MAJORITY：大多数的子规则成立就满足策略。

特别说明 ImplicitMetaPolicy\_MAJORITY 需要满足子规则数的计算方法：

```
threshold = len(subPolicies)/2 + 1
```

比如一共有 3 个子策略，需要至少 2 个子策略成立才能满足策略。如果总共有 4 个子策略，需要至少 3 个子策略成立才能满足策略。如果没有子策略，默认是满足的。

策略的内容可以有多种，下面分别来看几种策略：交易背书策略、链码实例化策略和通道管理策略。

### 3.5.2 交易背书策略

交易背书策略是对交易进行背书的规则，是跟通道和链码相关的，在链码实例化的时候指定。在链码调用的时候，需要从背书节点收集足够的签名背书，只有通过背书策略的交易才是有效的。这是通过应用程序和背书节点之间的交互来完成的，这在前面的交易流程里已经介绍过了。

#### 1. 交易背书策略的验证

背书是由一组签名组成的，每个 Peer 节点接收到区块时，都能根据交易的内容本地验证背书是否符合背书策略，不需要和其他节点交互。验证交易背书的基本原则是：

- 1) 所有的背书都是有效的，验证消息用有效的证书进行正确的签名；
- 2) 满足背书策略的有效背书数量，转化为 NOutOf 格式进行比较；
- 3) 背书是期望的背书节点签名的，在背书策略中指定了哪些组织和角色是有效的背书



节点。

如何来实现这几个原则的呢？我们先从背书签名的命令行语法开始，背书签名的语法 AND 和 OR 都可以转为 NOutOf:

□ AND(A, B) 可以转换为 NOutOf(1, A, B);

□ OR(A, B) 可以转换为 NOutOf(2, A, B)。

我们主要来看下 NOutOf 如何实现，背书策略的定义如下：

```
type SignaturePolicyEnvelope struct {
    Version    int32           // 背书策略版本，默认都是 0
    Rule       *SignaturePolicy // 背书策略规则：签名策略
    Identities []*common1.MSPPrincipal // 背书策略主体：MSP 主体签名
}
```

其中，MSP 主体 (Principal) 是基于 MSP 的身份标识的，有如下几种类型。

- 1) MSPPrincipal\_ROLE: 基于 MSP 角色的验证方法，目前只有 admin 和 member 两种。
- 2) MSPPrincipal\_ORGANIZATION\_UNIT: 基于部门的验证方法，同一个 MSP 中的不同部门。
- 3) MSPPrincipal\_IDENTITY: 基于某个具体身份证书的验证方法，验证签名是否有效。

MSPPrincipal 的定义如下：

```
type MSPPrincipal struct {
    PrincipalClassification MSPPrincipal_Classification // MSP 的类型
    Principal []byte           // 根据 MSP 的类型不同，实体有不同的内容
}
```

根据不同的 MSP 类型，主体是不同的。

#### (1) 基于 MSP 角色的验证

当 PrincipalClassification 是 MSPPrincipal\_ROLE 时，主体存储的内容如下：

```
type MSPRole struct {
    // MSP 标识符
    MspIdentifier string
    // MSP 角色：可选值是 MSPRole_MEMBER 和 MSPRole_ADMIN
    Role MSPRole_MSPRoleType
}
```

不同角色的验证方法如下：

- 1) MSPRole\_MEMBER: 验证是否为同一个 MSP 的有效签名；
- 2) MSPRole\_ADMIN: 验证签名者是否是 MSP 设置好的 admin 成员。

#### (2) 基于部门的验证

当 PrincipalClassification 是 MSPPrincipal\_ORGANIZATION\_UNIT 时，主体存储的内容如下：

```

type OrganizationUnit struct {
    // MSP 标识符
    MspIdentifier string
    // 组织部门标识符
    OrganizationalUnitIdentifier string
    // 证书标识符: 信任证书链和组织部门信息的哈希
    CertifiersIdentifier []byte
}

```

验证过程的步骤是:

- ❑ 验证是否为相同的 MSP;
- ❑ 验证是否是有效的证书;
- ❑ 验证组织部门信息是否匹配。

### (3) 基于身份证书的验证

当 PrincipalClassification 是 MSPPrincipal\_IDENTITY 时, 主体存储的内容如下:

```

type identity struct {
    // 身份标识符, 包含 MSP 标识符和身份编号
    id *IdentityIdentifier
    // 身份的数字证书, 包含了对公钥的签名
    cert *x509.Certificate
    // 身份的公钥
    pk bccsp.Key
    // 身份的 MSP 信息
    msp *bccspmsp
}

```

这样验证 MSP 是否是有效证书就可以了。

## 2. 命令行的背书策略语法

在命令行里, 可以用一种简单的语言, 根据主体的布尔表达式来表示策略。主体是用 MSP 来表示的, 用来验证签名者的标识和签名者在 MSP 里的角色。目前支持两种角色: member 和 admin。主体的表示方法是 MSP.ROLE, 其中 MSP 是 MSP 的标识, ROLE 可以是 member 也可以是 admin。这都是有效的主体: Org0.admin 表示由 MSP 标识 Org0 的任何一个管理员, Org1.member 表示由 MSP 标识 Org1 的任何一个成员。

其语法是:  $\text{EXPR}(\text{E}, \text{E} \dots)$ , 其中 EXPR 可以是 AND 也可以是 OR, E 可以为一个主体, 也可以为嵌套的 EXPR。比如:

1)  $\text{AND}(\text{'Org1.member'}, \text{'Org2.member'}, \text{'Org3.member'})$  要求 3 个 MSP 标识 Org1、Org2 和 Org3, 其中每个 MSP 都有 1 个成员有 1 个签名;

2)  $\text{OR}(\text{'Org1.member'}, \text{'Org2.member'})$  要求 2 个 MSP 标识 Org1、Org2, 其中任何 1 个成员有 1 个签名;

3)  $\text{OR}(\text{'Org1.member'}, \text{AND}(\text{'Org2.member'}, \text{'Org3.member'}))$  要求 MSP 标识 Org1 的成员有 1 个签名, 或者 MSP 标识 Org2 和 Org3 的成员都有 1 个签名。

目前在命令行的语法中，背书策略只支持 AND 和 OR 两种，并不支持更为复杂的 NOutOf。这部分的设计在后续内容中也会有调整。SDK 对背书策略都会转换成 NOutOf 语法，不过不是所有的 SDK 都支持。比如目前 fabric-sdk-go 提供的默认接口不支持 NOutOf 语法，但其内部是支持的，稍加改动很容易就能支持。详细可以参考 `cauthdsl_builder.go` 文件。

### 3. 给链码指定背书策略

背书策略可以在部署的时候用 -P 参数指定，后面是具体的背书策略。比如：

```
peer chaincode deploy -C testchainid -n mycc -p github.com/hyperledger/fabric/
examples/chaincode/go/chaincode_example02 -c '{"Args":["init","a","100","b","200"]}'
-P "AND('Org1.member', 'Org2.member')"
```

这个命令在链 `testchainid` 上部署链码 `mycc`，背书策略是 `AND('Org1.member', 'Org2.member')`。如果命令行里没有指定策略，那么默认的背书策略要求 MSP 标识 DEFAULT 成员的一个签名。

#### 3.5.3 链码实例化策略

链码实例化策略是用来验证是否有权进行链码实例化和链码升级的。链码实例化策略是在对链码打包和签名的时候指定的，如果没有指定实例化策略，默认是通道的管理员才能实例化。

```
type SignedChaincodeDeploymentSpec struct {
    // 链码部署规范
    ChaincodeDeploymentSpec []byte
    // 链码的实例化策略，结构同背书策略，在实例化的时候验证
    InstantiationPolicy []byte
    // 链码所有者的签名背书列表
    OwnerEndorsements []*Endorsement
}
```

链码实例化策略的定义和背书策略完全一样，验证方法也相同，只是用途和用法不一样。链码实例化策略是直接从链码打包中获取的，实例化完成后会将策略存放在链上。在链码实例化和升级的时候会先验证是否符合当前的实例化策略，验证通过才可以更新链码实例化策略。存储在链上的链码信息结构如下所示：

```
type ChaincodeData struct {
    // 链码名称
    Name string
    // 链码版本
    Version string
    // 链码的 ESCC
    Escc string
    // 链码的 VSCC
    Vscv string
    // 链码的背书策略
```

```

Policy []byte
// 链码的内容: 包含链码的名称、版本、链码源码哈希、链码名称和版本的元数据哈希等内容
// 不包含链码源码
Data []byte
// 链码指纹标识, 目前没有使用
Id []byte
// 链码实例化策略
InstantiationPolicy []byte
}

```

链码信息结构 ChaincodeData 在链上是按链码的名称索引的。

### 3.5.4 通道管理策略

通道配置是递归定义的:

```

type ConfigGroup struct {
    Version    uint64           // 配置版本
    Groups     map[string]*ConfigGroup // 子配置
    Values     map[string]*ConfigValue // 配置值
    Policies   map[string]*ConfigPolicy // 配置策略定义
    ModPolicy  string           // 配置修改策略的名称
}

```

其中, 配置值 ConfigValue 定义的是一些配置数据, 定义如下:

```

type ConfigValue struct {
    Version    uint64 // 配置版本
    Value      []byte // 配置数据, 可以是 JSON 结构的
    ModPolicy  string // 配置修改策略名称
}

```

比如在通道配置中区块生成间隔 BatchTimeout 设置的值是 “2s”, 局部的格式如下:

```

"BatchTimeout": {
  "mod_policy": "Admins",
  "value": {
    "timeout": "2s"
  }
}

```

我们再来看最重要的配置策略的定义:

```

type ConfigPolicy struct {
    Version    uint64 // 配置策略版本
    Policy     *Policy // 配置策略的内容, 这在前面已经介绍过
    ModPolicy  string // 配置策略中修改策略的名称
}

```

从上面的定义中我们可以看到, 配置策略是基于 SignaturePolicy 和 ImplicitMetaPolicy 的, ModPolicy 代表的是修改同级策略用到的策略名称。通道定义了 3 种配置策略, 如

表 3-1 所示。

表 3-1 3 种配置策略

策略名称	策略含义	策略说明
Readers	通道读取权限	任何读取通道交易的权限控制策略。比如定义哪些身份或者组织可以读取链上的数据，也包括调用排序服务的 Deliver 接口的权限和接收通道内事件的权限
Writers	通道写入权限	任何向通道提交交易的权限控制策略。比如定义哪些身份或者组织可以向链上提交交易，控制可以通过背书节点提交交易提案的权限
Admins	通道管理权限	任何修改通道配置的权限管理策略。比如定义哪些身份或者组织可以有通道配置的管理员权限，它指定了修改通道时需要的管理员签名

1. 通道配置的递归定义

我们来看一个简化的通道配置是如何递归定义的。

```
Channel:
  Policies:
    Readers
    Writers
    Admins
  Groups:
    Orderer:
      Policies:
        Readers
        Writers
        Admins
      Groups:
        OrdereringOrganization1:
          Policies:
            Readers
            Writers
            Admins
        Application:
          Policies:
            Readers
            Writers
            Admins
        ----->
      Groups:
        ApplicationOrganization1:
          Policies:
            Readers
            Writers
            Admins
        ApplicationOrganization2:
          Policies:
            Readers
            Writers
            Admins
```

在上面的配置中，最外层是 Channel，它定义了通道的子配置和策略定义。Channel 的

子配置里面定义了 Orderer 和 Application 配置，它们分别是相同的递归定义结构。其中 "--->" 显示的一行按照层级展开，代表的是 /Channel/Application/Writers。

怎么来使用这些配置策略呢？比如在排序服务节点调用 Deliver 接口的时候会检查这个节点是否满足 /Channel/Readers 策略。Peer 节点同步区块的时候也会检查是否满足 /Channel/Application/Readers 策略。

更详细的例子参考第 6 章的相关内容。

## 2. 通道配置的默认策略

在使用 configtxgen 工具生成创世区块或者通道配置时，使用的默认策略如表 3-2 所示。

表 3-2 通道配置的默认策略

策略路径	策略类型	策略定义
/Channel/Readers	ImplicitMetaPolicy	满足 /Channel/*/Readers 的 ANY 策略
/Channel/Writers	ImplicitMetaPolicy	满足 /Channel/*/Writers 的 ANY 策略
/Channel/Admins	ImplicitMetaPolicy	满足 /Channel/*/Admins 的 MAJORITY 策略
/Channel/Application/Readers	ImplicitMetaPolicy	满足 /Channel/*/Readers 的 ANY 策略
/Channel/Application/Writers	ImplicitMetaPolicy	满足 /Channel/*/Writers 的 ANY 策略
/Channel/Application/Admins	ImplicitMetaPolicy	满足 /Channel/*/Admins 的 MAJORITY 策略
/Channel/Orderer/Readers	ImplicitMetaPolicy	满足 /Channel/*/Readers 的 ANY 策略
/Channel/Orderer/Writers	ImplicitMetaPolicy	满足 /Channel/*/Writers 的 ANY 策略
/Channel/Orderer/Admins	ImplicitMetaPolicy	满足 /Channel/*/Admins 的 MAJORITY 策略
/Channel/*/Org/Readers	SignaturePolicy	满足任意组织 Member 成员的有效签名，* 代表 Orderer 或者 Application，Org 是组织名称
/Channel/*/Org/Writers	SignaturePolicy	满足任意组织 Member 成员的有效签名，* 代表 Orderer 或者 Application，Org 是组织名称
/Channel/*/Org/Admins	SignaturePolicy	满足任意组织 Admin 成员的有效签名，* 代表 Orderer 或者 Application，Org 是组织名称

## 3.6 本章小结

本章从逻辑结构、节点结构、典型交易流程、消息协议结构、策略管理等几个方面介绍了 Hyperledger Fabric 1.0 的架构。通过这些内容的介绍，能够基本了解 Hyperledger Fabric 1.0 的设计原则和思路。



## 基于 Gossip 的 P2P 数据分发

### 4.1 概述

背书节点模拟执行签名的结果会经过排序服务 (Ordering Service) 广播给所有的节点, 它提供的是一种原子广播服务 (Atomic Broadcast), 即在逻辑上所有节点接收到消息的顺序是相同的, 相同序号都是相同的内容, 排序服务的详细原理和实现请参考第 6 章。排序服务广播的信息包括更新的状态信息和账本信息等, 这些信息需要广播给所有节点。如果排序服务和所有节点都保持直接连接, 在节点较多、数据量较大的情况下, 容易形成单点故障或成为性能瓶颈。

由超级账本节点组成的区块链网络本身就是一种去中心化的网络, 利用 P2P 实现数据广播是显而易见的做法, 最常见的实现方法是洪泛 (Flooding)。洪泛是节点在接收到数据包以后直接转发给所有的邻居节点, 直到所有的节点都接收到了数据包或者数据包的跳数 (Hop Count) 超过一定的限制。洪泛有很多优点, 比如节点覆盖度高。如果在一个源节点和目标节点之间存在一条路径, 洪泛就能通过广播找到这条路径, 并且能以最快的速度找到这条路径。洪泛还有很好的冗余度, 这在不稳定的网络中能提高网络的健壮性 (Robustness)。

洪泛的一个缺点是非常低效, 可能会出现广播风暴。超级账本采用基于 Gossip 的协议实现 P2P 的数据分发, 与洪泛 (Flooding) 的广播策略不同, 节点在接收到数据包以后, 不是直接转发给邻居节点, 而是会计算一下概率, 根据计算结果来判断是否需要进行转发。转发概率设置为固定值的纯 Gossip (Pure Gossip)、盲 Gossip (Blind Gossiping) 或者固定概率 Gossip (Fixed Probability Gossip)。转发概率还可以根据其他一些信息动态计算, 比如节点的度 (Degree)、全局的拓扑结构等。在超级账本的实现中, 采用的是随机的选择  $k$  (默

认值是3)个节点进行转发,如果邻居节点的数量还没有需要转发的节点数量多,就全部转发。

采用基于 Gossip 的广播策略,除了能提高转发效率外,还有以下一些好处。

1) **扩展性 (Scalability)**: 在网络节点数增加的情况下,整个系统的性能不会快速恶化。每个节点选择转发给邻居节点的数量和网络中的总节点数量是没有直接关系的。转发概率是基于本地信息进行计算的,这些信息的变化若较小,整个系统的性能就会比较稳定,系统就是可扩展的。

2) **适应性 (Adaptability)**: 在超级账本中,节点定期会跟其他节点交换信息。如果在这个过程中有节点发生故障,则会从存活节点中删除这个节点的信息。对于故障节点,还会定时检查是否已经恢复,恢复存活的节点会更新到存活节点列表中。如果有新加入的节点,也能通过节点信息的交换获取到,添加到存活节点列表中,广播给其他节点。这些都能通过 Gossip 协议学习到,自动调整网络的拓扑结构,适应网络节点的变化,保证整个网络的正常运行。

3) **优雅降级 (Graceful Degradation)**: 在可靠的广播协议中,存在一个值 $f$ ,如果错误数量没有超过这个值,则整个协议能正常运行;如果超过这个值,就会出现一些异常,或者完全不能工作。协议的可靠性就等于不超过 $f$ 个错误的概率。若想计算这个概率可能是非常困难的,计算概率所需要的数据也可能非常难以测量。优雅降级 (Graceful Degradation) 就是指协议能正确工作的概率不会因为错误数超过 $f$ 时就快速地降低。

## 4.2 超级账本中的 Gossip 协议

超级账本的 Peer 节点组成了一个 P2P 的网络,客户端 SDK (Go、Java、Python、Node.js 等不同语言) 会提交请求给 Peer 节点,Peer 节点处理后会提交交易提案 (Transaction Proposal) 给背书节点 (Endorser), 然后进行背书签名 (Endorsement), 最后经过排序服务达成共识后广播给 Peer 节点,如图 4-1 所示。Gossip 模块负责连接排序服务和 Peer 节点上,实现从单个源节点到所有节点高效的数据分发,在后台实现不同节点间的状态同步,并且可以处理拜占庭问题、动态的节点增加和网络分区。账本信息、状态信息、成员信息等都会通过 Gossip 协议进行分发。概括起来, Gossip 协议主要完成的功能和目标有以下几个。

1) 在不需要所有节点都连接到排序服务获取账本区块数据的情况下,超级账本网络中所有节点还能有相同的账本数据、状态信息。

2) 系统已经正常运行后,对于新加入到超级账本网络中的节点,可以不直接连接到排序服务就能从网络中其他节点处获取到账本数据、状态信息。

3) 那些错过了批量更新的节点 (比如由于网络中断或者临时的超负荷运行没有接收到数据),能够保证落后的节点获取到缺失的区块。

4) 维护和管理成员信息,跟踪哪些成员是存活的,哪些是有故障的。

5) 数据能快速地从一个源节点同步到所有的其他节点上, 能够保证大量的数据可以在节点之间进行传输。

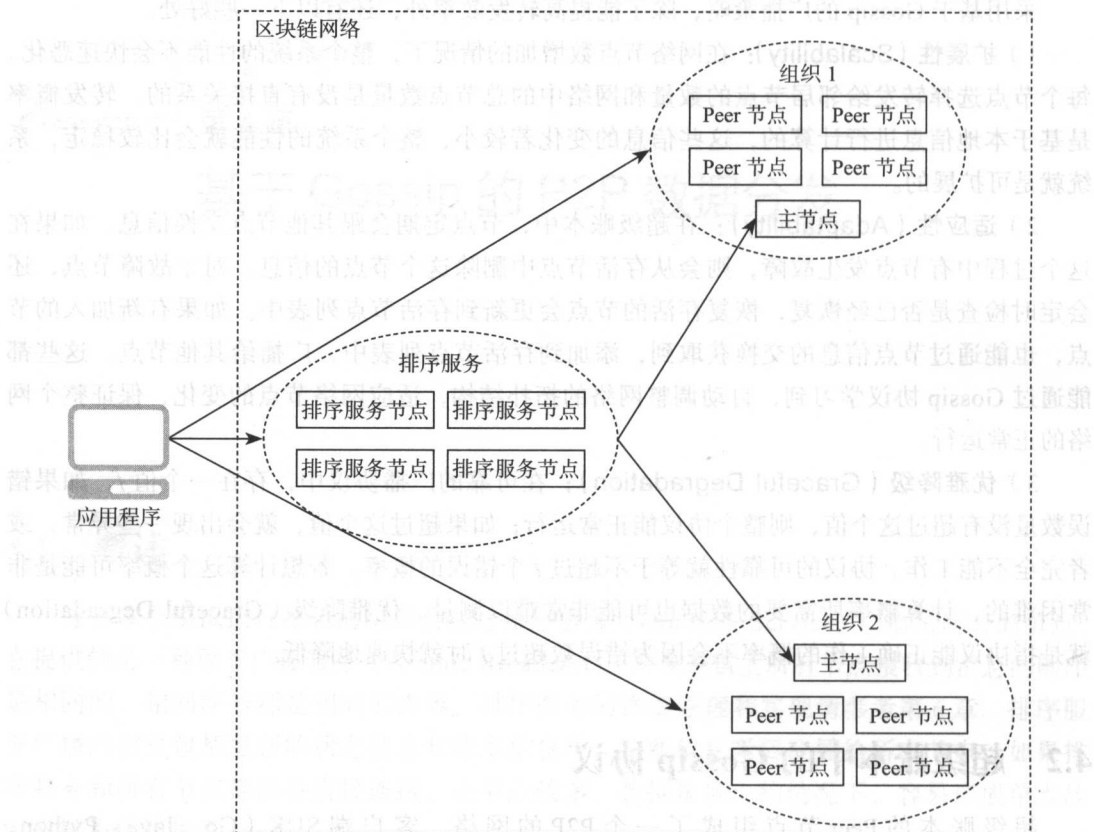


图 4-1 基于 Gossip 的通信路径

超级账本网络会通过主节点选举 (Leader election) 选择一些节点, 通过调用 Deliver() 接口连接到排序服务, 这些节点负责把接收到的批量区块 (batch) 广播给其他节点。每个组织都会根据自己的需要选择一个节点来连接排序服务, 再把批量区块分发给组织内的其他节点。后面会专门阐述主节点选举的过程。

基于 Gossip 的广播实现的过程是, 一个 Peer 节点接收到消息以后, 随机地选择  $k$  个节点, 把消息发送给它们。如果有节点没有响应, 则同步更新发现模块的数据。另外, 还有一个基于反熵 (Anti-entropy) 的状态同步过程, 它会在不同节点间同步状态。它们周期性地和其他节点比较信息, 保持状态的同步。每个节点都会维护完整的成员信息, 这样它们可以在发送信息的时候自行随机地选择节点。在这种模式下, 不需要维护固定的连接, 因为它是非常健壮的, 相对来说这也是非常容易实现的, 同时还能处理节点故障和拜占庭问题。

### 4.3 成员认证及身份管理

在 Gossip 网络中, 每个节点都有超级账本网络认可的 MSP (Membership Service Provider) 颁发的证书 (身份, Identity), 从证书计算哈希值导出 (目前的实现是直接 Endpoint 转换而来的) 的一个标识符称为节点的 PKI-ID。

身份管理模块管理节点标识符和证书之间的映射, 在内部维护了以 PKI-ID 为键、证书为值的映射表 `pkiID2Cert`, 可以通过 PKI-ID 获取节点的证书, 也可以更新节点的证书。同时还内置了 MCS (MessageCryptoService) 模块, 它可以对消息进行签名和验证。更新节点证书的时候也会通过 MCS 模块验证证书是否有效, 检查从证书导出的标识符是否和 PKI-ID 匹配。

在生产环境中, 假设所有节点都是双向 TLS 部署的, TLS 连接的双方都有一个有效的 TLS 证书。当节点和对端节点第一次连接时, 会有一个握手协议, 这个协议验证它是否拥有 TLS 证书的私钥, 因而在 TLS 会话中绑定成员身份。握手是相互的, 过程比较简单, 握手节点主动发送给对端节点一条 `ConnEstablish` 消息, 包含的内容有:

- 节点 TLS 证书的哈希值;

- 节点的 PKI-ID;

- MSP 身份证书。

然后, 对端节点的处理过程是:

- 接收发送过来的 `ConnEstablish` 消息;

- 提取节点的 TLS 证书并计算哈希值;

- 通过哈希值验证 `ConnEstablish` 的签名是否正确。

如果签名验证通过, 还会检查 PKI-ID 是否在黑名单列表中, 如果不在黑名单中, 就信任这个节点。用 PKI-ID 作为键值, 保存证书到映射表 `pkiID2Cert` 中, 后续可以验证这个节点发送消息的签名。同时还会用 PKI-ID 作为键值, 创建一个连接并进行关联, 并将其保存在内存中。需要和这个节点通信的时候通过 PKI-ID 快速地找到连接并发送消息。发送的过程是把消息放到一个发送缓冲区中, 由一个线程负责把发送缓冲区中的数据发送出去。如果签名验证失败, 节点拒绝这次连接。

### 4.4 节点启动及成员管理

超级账本网络维护着所有节点的信息, 包括存活节点和故障节点, 包括最近一次检测到它们存活或者掉线的时间, 所有节点信息都是以 PKI-ID 为标识符的。

要想加入到超级账本网络, 节点必须至少要知道网络中一个存活节点的地址信息。节点启动的时候会读取配置文件 `core.yaml`, 读取 `bootpeer.gossip.bootstrap` 字段的值, 这个字段可以设置为一个列表, 它包含了它可以连接的一些节点, 这个列表名为启动集合

(Bootstrap Set)。节点会给启动集合中的所有节点发送 **MembershipRequest** 消息，其包含的内容是：

1) **AliveMessage** 消息；

2) 本地节点已知的存活节点列表。

其中，**AliveMessage** 消息包含的内容有以下几项。

1) **PKI-ID**；

2) **Endpoint** (host + “:” + port)；

3) **Metadata** (字节数组，以后备用)；

4) **PeerTime**，它又包含以下内容：

□ 节点转世时间 (即启动时间)；

□ 单调递增的计数器，每次在 **AliveMessage** 传播时加 1。

5) 上面所有字段的签名；

6) 节点的证书 (可选)。

当一个节点接收到其他节点发送过来的 **AliveMessage** 后，首先会调用 **MCS** (**MessageCryptoService**) 模块验证消息的证书和签名，若验证通过，会添加到存活节点列表中，并更新检测到存活的时间。

节点证书只在节点启动后的一段时间内随消息发送，过了这段时间后消息就不再包含证书信息。当一个节点接收到其他节点的消息时，它会用之前收到的这个节点发送的证书进行验证。这就是为什么在节点启动后的一段时间内在 **AliveMessage** 中需要包含证书信息，就是为了在没有证书的情况下能够验证这个节点签名的信息。没有接收到其他节点证书的节点，可以通过周期性的数据交换机制来获取缺失的证书。

节点启动以后，每个节点还会周期性地在网络中广播 **AliveMessage** 消息，接收到消息的节点会更新本地的存活节点列表。每个节点根据接收到的存活消息判断故障节点。这是最简单的处理拜占庭问题的办法，这里考虑到存活消息是由节点签名的，因而不能被攻击者伪造，其他节点也不能通过发送消息就改变成员的状态。同时每个节点还会周期性地检查本地的存活节点列表，查看是否有节点在相当长的一段时间内没有更新存活状态，如果有节点掉线了，就从存活列表中删除它，添加到掉线节点列表中。节点会周期性地尝试重新连接掉列表中的节点。

## 4.5 主节点选举过程

主节点选举 (Leader election) 的用处是，判断在相同组织 (Organization) 中哪个节点可以作为代表连接排序服务。主节点选举过程是在 **Gossip** 层实现的，而且假设在某个时间段内，同时存在多个主节点，以避免拜占庭行为发生。

主节点选择过程是基于组织范围 (Scope) 内广播的 **LeadershipMessage** 消息的，主节点



选举的消息有两种类型。

1) 一种是参与主节点选举的消息 `proposal`，它在竞争主节点的过程中广播给所有节点的消息。

2) 一种声明成为主节点的消息 `declaration`，它通过比较，可申请为主节点的消息。

两种消息的结构是一样的，包含的内容如下所示：

1) 节点的 `PKI-ID`。

2) 是否声明为主节点。

3) `PeerTime`，它又包含以下内容：

□ 节点转世时间（即启动时间）；

□ 单调递增的计数器，每次发送一个 `LeadershipMessage` 消息时加 1。

当一个节点启动的时候，它先等待网络稳定再开始参与主节点选举。网络稳定的等待时间是 15 秒，如果在等待时间内节点数不再变化或者已经出现了主节点，这就是稳定了。如果在等待过程中没有产生主节点，则参与主节点的选举。参与主节点选举前设置自己的“`isLeader`”属性为 `false`，广播一个 `proposal` 消息给组织内其他的节点。然后节点会设置一个超时时间（5 秒），等待 `proposal` 消息在组内扩散，其他节点接收到 `proposal` 消息会暂存起来。若超时时间到了，还没有出现主节点，则按 `PKI-ID` 的字母顺序排序，最靠前的节点（也可以采用其他的算法）标识“`isLeader`”为 `true`，声明自己为主节点，在组织内广播一个 `declaration` 消息。

如果在时间超时（5 秒）之前有其他节点声明为主节点，则接受它为主节点。如果同时有多个节点声明为主节点，名称最靠前的节点会成为主节点，其他节点主动放弃，它们标记自己的“`isLeader`”属性为 `false`。所有的节点都认定在声明为主节点的 `Peer` 节点中名字最靠前的节点为主节点。如果主节点掉线，剩余节点里名字最靠前的声明为主节点。如果这时发现有另外一个名称更靠前的节点声明为主节点，则当前节点就会主动放弃。目标是最终只有一个节点设置“`isLeader`”为 `true`。

一次主节点选举的有效时间 `leaderAliveThreshold` 是 10s，没有成为主节点的节点清除收到的 `proposal` 消息，然后等待有效时间结束，进入下一轮主节点选举的过程。

## 4.6 基于反熵的状态同步

反熵（`Anti-entropy`）是指每个节点周期性地和邻居节点交换保存的数据，然后对比本地数据和邻居节点所保存的数据，检查是否有缺失或者过期的数据，然后更新本地节点的数据为最新的数据。

超级账本的反熵实现比较简单，每个节点定期（10s）检查本地账本的区块序列号和其他节点账本的序列号。若发现本地的序列号比网络中其他节点账本的序列号小就在网络中广播一个 `GossipMessage_StateRequest` 消息，请求缺失序列号的区块。收到请求的节点如果



有对应序列号的区块，会在网络中广播一个 `GossipMessage_StateResponse` 消息，使其包含本地账本中请求序列号的区块。这种方式是通过直接消息（`directMessage`）渠道进行的，节点接收到消息后会缓存起来，放到一个 `PayloadsBuffer` 的数据结构里保存起来。

由于 TCP/IP 网络传输的特点，数据可能不是按序到达的，`PayloadsBuffer` 会在内部保存一个索引，记录等待提交账本的下一个区块的序列号 `next`。如果接收到的区块序列号小于 `next`，说明是过期的区块，直接丢弃。如果序列号大于 `next`，`PayloadsBuffer` 把收到的数据放到序列号对应的缓冲区数组里。只要收到的数据和已提交区块序列号连续了，就会把连续的数据区块提交到账本里，然后删除缓冲区中已提交的数据区块，同时更新索引 `next`。

另外一个更新状态数据的过程是在主节点从排序服务中获取到区块以后，会创建一个类型为 `GossipMessage_DataMsg` 的数据消息广播给其他节点，其他节点接收到区块后同样也会和 `PayloadsBuffer` 中的 `next` 进行比较，进行同样的处理。和直接消息方式不同的地方是，从排序服务获取到的 `GossipMessage_DataMsg` 消息只包含一个区块，直接消息里可能会包含多个缺失的区块。

## 4.7 数据传播过程

数据传播机制（Data dissemination）的底层是由基于 Gossip 的节点通信支撑的。在这种模式下，不会构建每个节点之间的分离路径（disjoint path）信息，而是每个节点每次都随机性地选择  $k$  个节点来扩散消息。节点在某个时间点随机性地选择节点交换信息，信息流就在整个系统中流动起来了。这种交互方式比固定结构的方式更健壮，而且在遇到节点变动或者拜占庭问题的时候更容易维护。而且，固定结构的模式保留了每个节点之间的分离路径信息，消息复杂度会更大。注意，一个节点在随机选择其他节点时，它可以参考当前的成员视图、前一段时间的历史存活节点、启动集合，以及所有节点的列表等信息。

每个节点有更高的出度（Outgoing Degree），不同节点独立选择不同的传播路径，这能够保证当路由中有拜占庭节点的时候不会阻止整个网络接收特定的消息。允许一个节点转发消息给更多的节点，系统自然能在遇到拜占庭节点的时候有更好的健壮性，这样网络中传输的消息会更多。处理拜占庭错误的健壮性与每一轮选择消息的发送数量及节点数量的通信开销之间是需要权衡的。理论上的推导和结果模拟说明，每个节点以一个相对小的出度能够在高比例的拜占庭节点的情况下保证一个合适的扩散成功率。

## 4.8 多通道的支持

创建通道是为了限制信息传播的范围，是和某一个账本关联的。每个交易都是和唯一的通道关联的。这会明确地定义哪些实体（组织及其成员）会关注这个交易。

客户端 SDK 通过发送一个 CONFIGURATION 的交易背书（Endorsement）请求来创

建一个通道，然后通过排序服务广播给其他节点。创建通道的请求包含组成通道的组织 (Organization) 列表，即哪些组织可以加入到这个通道中。

一旦创建好了通道，客户端 SDK 就可以通知组织内的节点加入到新创建的通道中。节点的 Gossip 模块就会给这通道内的组织广播一个消息：它属于这个通道了。

Gossip 要在多通道环境下还能正常工作，成员管理需要对通道内的成员节点进行维护，也就是说通道内的所有节点都需要知道其他节点的存在。代表一个组织和排序服务进行连接的节点会接收到哪些节点属于这个通道，会根据需要调用 Deliver() 接口。排序服务能够保证代表一个组织参与到这个通道的节点能够接收到区块数据。连接节点会分发新收到的区块数据给加入到这个通道的节点。当成员管理建立起来后，Gossip 模块就会在隧道内部分发数据。不属于这个通道的数据项会在通道外传输，包括交易信息和与 Gossip 相关的会员管理信息。通道信息不会在通道外的节点间传播。

一旦一个节点加入到一个通道中，它需要获取最新的通道配置信息，用来识别参与的组织。节点加入通道的时候，客户端 SDK 就会提供通道最新的配置交易信息，里面会包含参与的组织。对于新创建的通道，这会是一个创世区块 (Genesis Block)；对于老的通道来说，这是一个最新的重新配置的块。

节点获取到这些信息后，Gossip 模块就会在通道允许的组织范围内分享这些信息。这样，通道的成员管理信息是在 Gossip 服务允许的范围内进行维护的。这些信息都建立起来以后，Gossip 就会基于这些信息进行工作，就像只有一个通道一样。

通道内 Gossip 的工作方式和标准的 Gossip 是一样的。需要注意的是，成员管理是每个通道独立的，因而所有的操作都在这个通道的成员管理范围内。特别地，一个节点可以选择通道内的一些节点来进行区块转发，再选通道内的其他一些节点来进行状态同步。注意，状态可能会在不同的组织之间同步，只要它们是同一个通道的成员。

从通道中删除一个组织的过程和创建一个通道是类似的。客户端 SDK 会发送一个 CONFIGURATION 的交易背书请求。通道内的 Peer 节点背书完成后，删除组织和成员的交易就会发送给排序服务节点。然后，其他节点会收到一个更正过组织列表的通道消息。每个收到消息的 Gossip 模块都会移除被删掉的节点，因而，和通道相关的信息只会在剩下的通道成员之间传输。

被通道删掉的 Peer 节点最终会发现自己被删除了。因为 Peer 节点的 Gossip 成员管理模块会继续发送心跳信息，但是这些信息不会发送给已删除的节点。所以，过了一段时间以后，节点就知道自己不再属于这个通道了。

## 4.9 消息的验证策略

每个通过 Gossip 协议转发给其他节点的消息都会声明节点的一些信息，其包括以下内容：

- 必须包含节点的 PKI-ID；

□ 必须由节点进行签名;

□ 能够通过节点的证书进行验证。

节点间点对点传播的消息没有签名,不会通过 Gossip 转发。假设在生产环境下,节点的 TLS 默认是开启的,并且有安全方面的考虑(防止流量劫持、重放攻击等)。唯一一个不用节点签名而且不通过点对点方式传播的消息是账本数据区块,它是由排序服务进行签名的。

Peer 节点接收到排序服务广播的数据区块以后,可以验证附加在区块里的签名信息,这个签名信息可以用来作为  $k/n$  的多签名( $n$  个节点中至少有  $k$  个签名验证通过)验证策略。比如,SOLO 和 Kafka 的排序服务只要求节点验证单个数字签名这样就可以验证分发的区块了。SBFT 则是  $n$  个里  $f+1$  的策略,要求每个区块都要验证  $f+1$  个排序节点的数字签名。其他的排序服务可以指定其他的验证策略或者不同的  $k$  值。

Gossip 模块初始化以后,它就会设置通道的验证策略,以判断节点所属的组织,并且它只给通道内的节点发送区块。这个过程可以通过关联每个节点的 PKI-ID 和其组织的根证书来实现,本地账本都有通道里组织的最新配置。

由排序服务广播的批块(Batch)包含了通道中最新配置区块(Configuration block)的序列号。当一个节点接收到其他节点发送过来的区块时,会检查区块的序列号和自己本地账本的序列号,然后就可以知道是不是可以安全地转发这个区块了。检查方法是查看最新配置区块的序列号,如果收到的数据区块序列号比提交到账本的最新序列号高,则数据区块就会缓存到内存中,不会转发给其他节点。否则,账本的最新配置就是最新的,区块就可以安全地转发给通过策略验证的节点了。

超级账本声明了一个内部消息的存储接口 MessageStore:

```
type MessageStore interface {
    // 添加消息 msg 到内存中, 返回是否添加成功
    Add(msg interface{}) bool

    // 返回存储的消息数量
    Size() int

    // 返回存储的所有消息
    Get() []interface{}
}
```

具体的实现如下所示:

```
type messageStoreImpl struct {
    pol      common.MessageReplacingPolicy
    lock     *sync.RWMutex
    messages []*msg
    invTrigger invalidationTrigger
}
```

包含一个通用的消息验证策略函数：

```
type MessageReplacingPolicy func(this interface{}, that interface{})
InvalidationResult
```

其中，this 和 that 指代的是当前消息和原有消息的比较，并判断当前消息的有效性。比较的结果 InvalidationResult 有 3 种可能。

1) MESSAGE\_INVALIDATES：当前消息是有效的，原有消息是无效的，用当前消息替换原有消息；

2) MESSAGE\_INVALIDATED：当前消息是无效的，原有消息是有效的，丢弃当前的消息；

3) MESSAGE\_NO\_ACTION：两个消息可能是不同类型的，两个消息不进行比较，两个消息都是有效的。

MessageReplacingPolicy 可以根据实际存储的消息定义不同的比较策略。

invalidationTrigger 是一个回调函数，当替换原有消息时，可以对替换掉的消息进行一些处理。

我们来看几种具体消息类型实现的验证策略。目前用到的内部消息类型有：存活消息 (Alive)、区块数据 (Data)、状态消息 (State)、身份信息 (Identity)、主节点选举 (Leader) 消息。每个消息都有一个时间戳信息 PeerTime，它包含两个字段 incNumber 和 seqNum。判断两个消息的有效性和消息类型有关，验证方法可以分为 3 类。

1) 基于时间戳的比较。总的原则是 incNumber 大的消息有效，有相同 incNumber 的消息，seqNum 大的有效，incNumber 和 seqNum 都相同的话以原有消息为准，当前消息若是无效消息，直接丢弃。

2) 基于消息序列号的比较。这种比较策略只对数据区块进行比较，相同 seqNum 的还会比较数据哈希值，用相同哈希值的替换原有消息，否则两个消息都是有效的。若 seqNum 不同则要检查缓冲区大小能否存储两个消息 seqNum 之间的所有消息。如果缓冲区足够存放，则两个消息都是有效的，否则 seqNum 大的消息有效。

3) 基于节点 PKI-ID 的比较。检查发送两个消息的节点 PKI-ID 是否相同，如果相同就以当前消息为准，已有的消息就过期无效了。这种验证方法只用在身份信息中，每个节点定期都会广播身份信息，其他节点就会以最后收到的信息为准，作为相同 PKI-ID 的节点身份信息。

不同消息类型的验证策略如表 4-1 所示。

表 4-1 不同消息类型的验证策略

消息类型	消息验证策略
存活消息	基于时间戳比较
区块数据	基于消息序列号比较
状态消息	基于时间戳比较
身份信息	基于节点 PKI-ID 比较
主节点选举消息	基于时间戳比较

## 4.10 消息的多路分用及分区

通过 Gossip 协议广播的消息种类较多，不同种类的消息有不同的处理逻辑。Gossip 模块利用 GO 语言的通道，实现一个消息的多路分用接

## □ ChannelDeMultiplexer:

```
type ChannelDeMultiplexer struct {
    channels []*channel
    lock      *sync.RWMutex
    closed    int32
}
```

其中 lock 是一个读写锁，用来同步对 channels 的处理；closed 是通道是否关闭的标识，通道关闭就不能再从通道中读取数据；channels 是一个 channel 数组，如下所示：

```
type channel struct {
    pred common.MessageAcceptor
    ch    chan interface{}
}
```

ch 是缓存消息的通道，默认只能保存 10 个消息。节点接收到 Gossip 消息的时候，会调用 DeMultiplex 接口。pred 用来判断订阅者对某个消息是否感兴趣，感兴趣的消息会过滤出来存放在 ch 里，等待下一步的处理。pre 的策略可以根据业务逻辑来实现，函数定义如下：

```
type MessageAcceptor func(interface{}) bool
```

我们来看看 Gossip 模块实现的几种 MessageAcceptor。

1) Gossip 消息过滤器。Gossip 模块接收到的消息类型有：GossipMessage 和 Signed GossipMessage，如果不是这两种类型的消息，会过滤出来丢弃。

2) 存活消息过滤器。过滤出类型为 GossipMessage\_AliveMsg、GossipMessage\_MemReq 和 GossipMessage\_MemRes 的消息。

3) 状态消息过滤器。过滤出状态同步消息。

4) 远程状态消息过滤器。过滤出 GossipMessage\_StateRequest 和 GossipMessage\_StateResponse 的消息，这两种消息是为了状态同步而主动发送和接收的消息。如果它包含了连接验证信息，则会调用 MCS 验证是否是指定通道成员发送的消息。

5) 主节点选举消息过滤器。过滤出某个通道上的 GossipMessage\_LeadershipMsg 消息。

我们来看看 Gossip 模块中的多路分用消息过滤器漏斗，如图 4-2 所示。

节点的 Gossip 模块会绑定 gRPC 端口（默认是 7051 端口），从其他节点或者排序服务节点上获取到的消息，通过 handler 调用 DeMultiplex 实现消息的多路分用，消息会通过消息过滤器过滤后进行业务逻辑的处理。这里的过滤器都虚线来表示，以说明同一层的过滤器不是互斥的，就是说同一层的过滤器输入都是相同的，经过过滤器的输出会不同。如果经过多层过滤器的过滤，消息是在上一层过滤的基础上筛选出来的。比如“状态数据过滤器”和“状态同步消息过滤器”的输入都是“状态消息过滤器”的输出，而“状态消息过滤器”的输入是 gRPC 消息过滤出的 Gossip 消息。

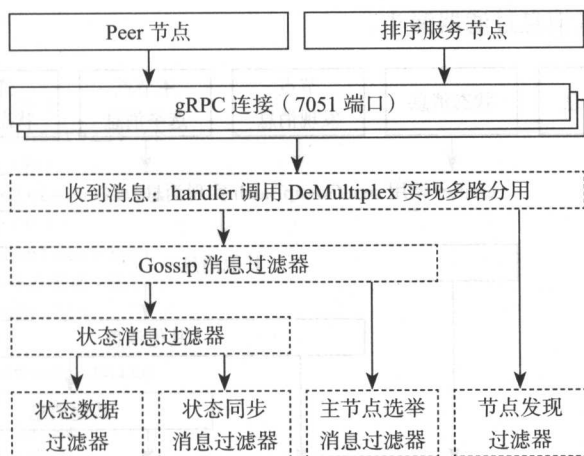


图 4-2 多路分用消息过滤器漏斗

ChannelDeMultiplexer 中的 channels 可以定义多个，同一个消息只要满足 pre 定义的过滤条件，就会放到对应的 ch 里，所以可能不同的 channel 里有相同的消息。对于需要广播的消息，Gossip 模块还实现了消息分区，同一个消息经过过滤器过滤以后，只会出现在一个列表中。函数实现比较简单，代码如下：

```

func partitionMessages(pred common.MessageAcceptor, a
*proto.SignedGossipMessage) ([]*proto.SignedGossipMessage,
[]*proto.SignedGossipMessage) {
    s1 := []*proto.SignedGossipMessage{}
    s2 := []*proto.SignedGossipMessage{}
    for _, m := range a {
        if pred(m) {
            s1 = append(s1, m)
        } else {
            s2 = append(s2, m)
        }
    }
    return s1, s2
}

```

经过 partitionMessages 处理过的消息 a，会分成两个切片，满足过滤器 pre 的放到切片 s1 中，不满足的放到切片 s2 中。

广播消息过滤器有以下几种：

- ❑ 区块数据过滤器：过滤出指定通道 chainID 的区块数据。
- ❑ 状态消息过滤器：过滤出状态同步消息。
- ❑ 通道内部的消息过滤器：过滤出只在通道内部广播的消息。
- ❑ 组织内部的消息过滤器：过滤出只在组织内部广播的消息。
- ❑ 主节点选举消息过滤器：过滤出某个通道上的 GossipMessage\_LeadershipMsg 消息。



图 4-3 所示为分区消息过滤器漏斗。

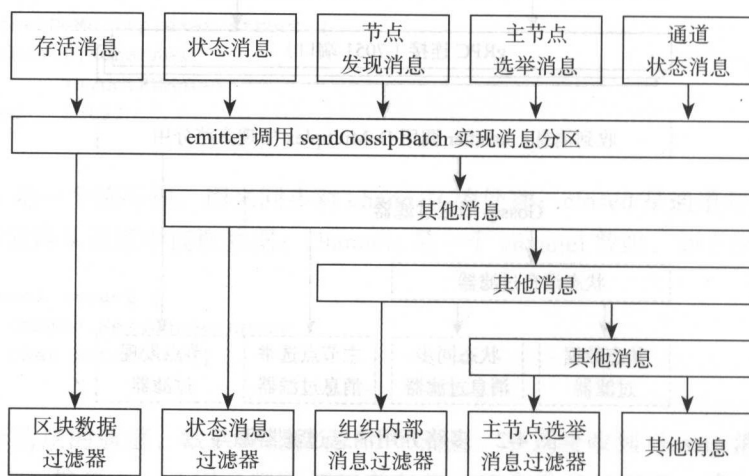


图 4-3 分区消息过滤器漏斗

和消息多路分用不同，经过分区以后消息只能在一个结果集中，同一层的过滤器是互斥的。比如经过“区块数据过滤器”过滤以后，“状态消息过滤器”等就只能在其他消息中进行筛选。

## 4.11 和 Gossip 相关的配置参数

和 Gossip 相关的配置参数如下所示：

```

# 启动连接节点，可以是多个
bootstrap: 127.0.0.1:7051
# 自动选择主节点还是指定主节点
useLeaderElection: false
# 只在 useLeaderElection 设置为 false 的时候生效
orgLeader: true
# 本地节点的 ID: ip:port
endpoint:
# 区块缓冲区大小
maxBlockCountToStore: 100
# 消息推送间隔(毫秒)
maxPropagationBurstLatency: 10ms
# 消息推送缓冲区大小
maxPropagationBurstSize: 10
# 消息推送次数
propagateIterations: 1
# 消息推送节点数
propagatePeerNum: 3
# 消息获取间隔(秒)

```

```

pullInterval: 4s
# 消息获取节点数
pullPeerNum: 3
# 状态消息获取间隔 (秒)
requestStateInfoInterval: 4s
# 状态消息推送间隔 (秒)
publishStateInfoInterval: 4s
# 状态消息存活周期 (秒)
stateInfoRetentionInterval:
# 存活消息中推送证书的周期 (秒)
publishCertPeriod: 10s
# 是否验证区块消息
skipBlockVerification: false
# 是否忽略安全检查
ignoreSecurity: false
# 建立连接超时时间 (秒)
dialTimeout: 3s
# 连接超时时间 (秒)
connTimeout: 2s
# 接收消息缓冲区大小
recvBuffSize: 20
# 发送消息缓冲区大小
sendBuffSize: 20
# 摘要消息处理超时时间 (秒)
digestWaitTime: 1s
# 请求消息处理超时时间 (秒)
requestWaitTime: 1s
# 消息响应超时时间 (秒)
responseWaitTime: 2s
# 存活消息间隔时间 (秒)
aliveTimeInterval: 5s
# 存活消息超时时间 (秒)
aliveExpirationTimeout: 25s
# 重新连接间隔时间 (秒)
reconnectInterval: 25s
# 外部标识
externalEndpoint:

```

## 4.12 本章小结

本章主要介绍由 Peer 节点组成的 P2P 网络如何实现数据的同步。区块链网络可能由较多的节点组成, Peer 节点和排序服务节点之间建立过多的连接容易造成网络异常和单点问题, 基于 Gossip 的 P2P 数据分发可以减轻排序服务节点的压力。

## 分布式账本存储

分布式账本技术 (DLT, Distributed ledger Technology) 还有一个名称叫共享账本 (Shared Ledger), 通过在不同节点之间达成共识, 记录相同的账本数据, 这是区块链技术的基础。本章讨论在 Hyperledger Fabric 1.0 中分布式账本技术的实现。

### 5.1 概述

超级账本采用背书 / 共识 (Endorsement/Consensus) 模型, 模拟执行和区块验证是在不同角色的节点中分开执行的。模拟执行是并发的, 这可以提高扩展性和吞吐量:

- ❑ 在背书节点 (Endorsing Peer) 处模拟执行链码 (Chaincode);
- ❑ 在所有的 Peer 节点上验证交易并提交。

每个 Peer 节点会维护多个账本, 如图 5-1 所示。

超级账本包含以下元素。

- ❑ 账本编号: 快速查询存在哪些账本;
- ❑ 账本数据: 实际的区块数据存储;
- ❑ 区块索引: 快速查询区块 / 交易;
- ❑ 状态数据: 最新的世界状态数据;
- ❑ 历史数据: 跟踪键的历史。

每个 Peer 节点会维护 4 个 DB, 它们分别是:

- ❑ idStore, 存储 chainID;
- ❑ stateDB, 存储 world state;

❑ versioned DB, 存储 key 的版本变化;

❑ blockdb, 存储 block。

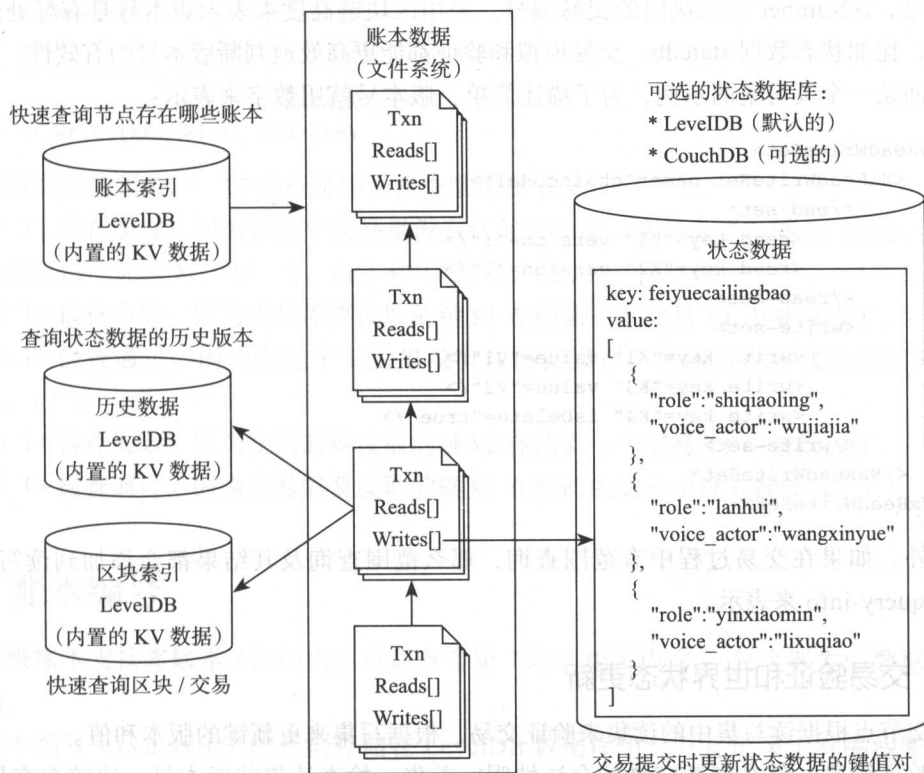


图 5-1 分布式账本存储

## 5.2 读写集

### 5.2.1 交易模拟和读写集

在背书节点 (Endorser) 模拟执行交易的过程中, 会生成读写集 (Read-Write Set)。读集 (Read Set) 包含了唯一键的列表, 还有在模拟执行过程中交易读取的已提交键值。写集 (Write Set) 也包含了一个唯一键的列表, 还有在模拟执行过程中交易写的键值。交易过程中有删除键, 它会记录删除标记。如果在一个交易中对同一个键进行了多次更新, 则会以最后一个为准。交易只会读取已提交的数据, 即使在交易中更新了某个键的值。就是说, 不支持读取本次交易更新后的结果。

读集会包含键的版本, 写集只会包含键的最新值。

版本号是使每个键不重复的唯一标识, 实现的方法有很多, 比如可以采用单调递增的数

值来表示。在目前的实现中，采用的是区块链的高度来表示，一个交易中所有键值修改的版本号都是相同的。高度是一个二元组  $\text{Heigh}(\text{blockNumber}, \text{txNuber})$ ，其中， $\text{blockNumber}$  是区块号， $\text{txNumber}$  是区块内的交易编号。采用区块链高度来表示版本号是有好处的，其他模块，比如状态数据  $\text{statedb}$ 、交易模拟和验证都能更高效地判断版本号的有效性。

下面是一个读写集的示例，为了描述简单，版本号就用数字来表示：

```
<TxReadWriteSet>
  <NsReadWriteSet name="chaincode1">
    <read-set>
      <read key="K1" version="1"/>
      <read key="K2" version="1"/>
    </read-set>
    <write-set>
      <write key="K1" value="V1"/>
      <write key="K3" value="V2"/>
      <write key="K4" isDelete="true"/>
    </write-set>
  </NsReadWriteSet>
</TxReadWriteSet>
```

另外，如果在交易过程中有范围查询，那么范围查询及其结果都会添加到读写集中，它们用  $\text{query-info}$  来表示。

## 5.2.2 交易验证和世界状态更新

提交节点根据读写集中的读集来验证交易，根据写集来更新键的版本和值。

在验证阶段，怎么判断交易的合法性呢？首先，检查读集的版本号，比较在交易中读集里每个键的版本号是否和世界状态（World State）键的版本号一致。然后，如果读写集包含了  $\text{query-info}$ ，则会检查  $\text{query-info}$  包含的键是否有变化，比如新增键、更新或者删除键。要比较在模拟阶段交易进行范围查询的结果是否和验证阶段范围查询的结果是一致的。这能够保证在提交过程中如果出现了幻影项（Phantom Item），交易就会标记为无效。注意幻影保护只实现了范围查询（即链码调用  $\text{GetStateByRange}$ ），没有实现其他查询（比如链码调用  $\text{GetQueryResult}$ ）。其他查询是有幻影风险的，只能用在不提交给排序服务的只读交易中，除非应用能够保证在模拟阶段和提交阶段结果的稳定性。

如果交易通过了上面的检查，那么提交节点会根据写集来更新世界状态。遍历写集中的每个键，更新世界状态里对应的键值和版本号。

## 5.2.3 模拟和验证示例

为了便于理解，来看一个例子。假设一个键值对在世界状态里用一个三元组来表示： $(k, \text{ver}, \text{val})$ ，其中，键  $k$  最新版本  $\text{ver}$  的值是  $\text{val}$ 。

有 5 个交易  $T1$ 、 $T2$ 、 $T3$ 、 $T4$ 、 $T5$ ，它们都基于同一个世界状态的快照进行模拟。下

面的片段说明的是每个交易的读写集的操作：

```
世界状态 : (k1,1,v1), (k2,1,v2), (k3,1,v3), (k4,1,v4), (k5,1,v5)
T1 -> Write(k1, v1'), Write(k2, v2')
T2 -> Read(k1), Write(k3, v3')
T3 -> Write(k2, v2'')
T4 -> Write(k2, v2'''), read(k2)
T5 -> Write(k6, v6'), read(k5)
```

假设交易按照 T1 ~ T5 进行排序，检查结果如下。

- 1) T1 检查通过。因为在这个交易里没有任何读操作。交易会更新键 k1 和 k2，更新后世界状态里的三元组为：(k1, 2, v1') 和 (k2, 2, v2')。
- 2) T2 检查失败。因为交易需要读取的键 k1 在前面一个交易 T1 中被修改了。
- 3) T3 检查通过。因为在这个交易里没有任何读操作。更新后世界状态的三元组为：(k2, 3, v2'')。
- 4) T4 检查失败。因为交易需要读取的键 k2 在前面一个交易 T1 中被修改了。
- 5) T5 检查通过。因为交易需要读取的键 k5 在前面的交易中没有被修改。

## 5.3 账本编号

超级账本支持多账本（详细内容可以参考第7章的相关内容），每个账本的数据是分开存储的。

账本编号（LedgerID）的数据存储在 LevelDB 数据库中，只是记录了有哪些账本，创建新的账本会检查是否有相同的账本编号存在，这保证了全局唯一性。账本编号库并不存储与区块相关的数据，这和后面的区块索引不一样。

## 5.4 账本数据

账本数据（Ledger）是以二进制文件的形式存储的，每个账本数据存储在不同的目录下。后面的内容都是在已经区分了账本的情况下再对数据进行查询的。基于文件系统的区块存储实现了如下功能接口。

### 1) 账本存储管理。

- 提交区块到账本（AddBlock）
- 获取区块链信息（GetBlockchainInfo）
- 获取区块数据（RetrieveBlocks）
- 关闭区块存储（Shutdown）

### 2) 索引管理：跟踪区块和交易保存在哪个文件。

- 根据哈希值获取区块（RetrieveBlockByHash）



- 根据区块编号获取区块 (RetrieveBlockByNumber)
  - 根据交易编号获取交易 (RetrieveTxByID)
  - 根据区块编号和交易编号获取交易 (RetrieveTxByBlockNumTranNum)
  - 根据交易编号获取区块 (RetrieveBlockByTxID)
  - 根据交易编号获取交易验证码 (RetrieveTxValidationCodeByTxID)
- 账本数据的所有操作都是通过区块文件管理器 (blockfileMgr) 实现的, 定义如下:

```
type blockfileMgr struct {
    rootDir      string           // 区块链中区块存储的根目录
    conf         *Conf           // 配置信息
    db           *leveldbhelper.DBHandle // 数据库指针
    index        index       // 区块索引接口
    cpInfo       *checkpointInfo // 区块检查点信息
    cpInfoCond   *sync.Cond      // 条件变量
    currentFileWriter *blockfileWriter // 当前写入区块文件的指针
    bcInfo       atomic.Value     // 区块链信息
}
```

区块文件管理器实现的功能分为几类。

1) 账本数据存储管理。

□ 确定文件存储在哪个目录;

□ 确定区块存储在哪个文件。

2) 检查点管理: 跟踪最新持久化存储的文件。

3) 索引管理: 跟踪区块和交易保存在哪个文件。

#### 5.4.1 账本数据存储

区块文件管理器创建的文件名以 "blockfile\_" 为前缀, 6 位数字为后缀, 后缀必须是从小到大连续的数字, 中间不能有缺失, 比如 blockfile\_000000、blockfile\_000001、blockfile\_000002 等。默认的区块文件大小上限为 64MB (目前这个大小在代码中是固定的, 以后可能会动态调整)。一个账本能保存的最大数据量大概有 61TB, 这应该能满足目前绝大多数的应用了。

记账节点 (Committer) 负责维护节点本地的账本, 通过 Gossip 模块从排序服务接收到区块以后, 区块添加到账本的过程如图 5-2 所示。

区块文件管理器维护一个当前写入区块文件的指针 `currentFileWriter`, 写入区块文件的数据包括两个部分, 一个是区块大小, 另一个是区块数据。写入区块文件中的区块大小和区块数据都是经过序列化处理的, 序列化过程见技术实现部分。如果写入区块大小和序列化后, 当前区块文件的大小超过设定值, 则会写入到下一个区块文件中。创建一个新的区块检查点信息 (见区块索引部分), 保存到数据库中。

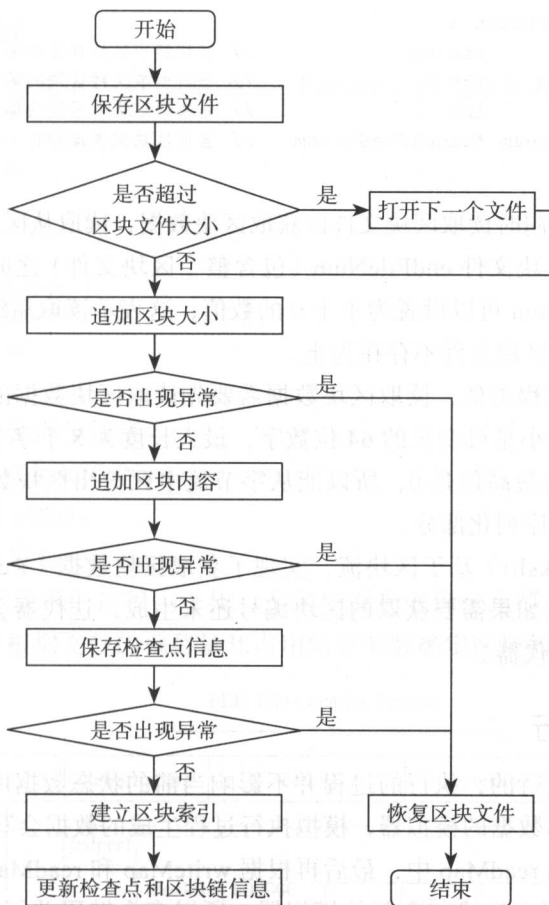


图 5-2 添加区块

若写入区块文件和保存区块检查点信息的过程出现任何异常，就会根据区块检查点记录的最新区块文件偏移 latestFileChunksize（见区块索引部分），恢复区块文件到写入前的状态。多个区块数据保存到区块文件后，区块文件形成了一个非结构化的二进制文件，它需要在区块文件外记录索引信息，才能快速地定位到区块。保存区块数据后会建立这个区块的索引，并保存到数据库中，详细的构建区块索引过程在后面的章节会有介绍。

最后是更新区块链文件管理器维护的区块检查点信息和区块链信息。

### 5.4.2 账本数据读取

区块文件流（blockfileStream）和区块流（blockStream）是以数据流的方式从区块文件系统中读取区块的，区块文件流是从单个文件中读取的，区块流可以跨不同的文件，实际的文件读取通过区块文件流来实现。区块流维护了当前的区块文件流指针，还有当前区块文件的编号等，详细的定义如下：

```

type blockStream struct {
    rootDir      string    // 区块链中区块存储的根目录
    currentFileNum int      // 当前读取文件区块的编号
    endFileNum    int      // 结束读取文件区块的编号
    currentFileStream *blockFileStream // 当前区块文件流指针
}

```

可以有多个区块流同时读取区块文件以获取区块数据，读取从区块文件 `currentFileNum` 的某个偏移开始，到区块文件 `endFileNum`（包含整个区块文件）之间的所有区块。结束读取文件区块的 `endFileNum` 可以设置为小于 0 的数值，这表示读取后续所有的区块文件，直到账本数据目录下某个区块文件不存在为止。

跟区块文件存储过程类似，读取区块数据需要先读取区块数据的大小，再读取区块数据本身。区块数据的大小是可变长的 64 位数字，最大长度为 8 个字节。由于可变长数字序列化后最后一个字节的最高位是 0，所以能从字节流中区分出区块数据大小和区块数据本身。详细的编码规则见序列化部分。

区块迭代器（`blocksIter`）基于区块流，实现了获取区块数据（`RetrieveBlocks`）的接口。增加的一个业务逻辑是如果需要获取的区块编号还未生成，迭代器会一直等待，直到获取到指定区块或者关闭迭代器。

### 5.4.3 交易模拟执行

链码是可以并行执行的，执行的过程并不影响当前的状态数据库。实现的方法是在最新账本上生成一个账本数据的模拟器，模拟执行过程生成的数据会写入模拟器的 `writeMap` 中，读取的数据写入到 `readMap` 中，最后再根据 `writeMap` 和 `readMap` 生成 `TxRwSet` 结果。每次链码执行的时候都会生成一个新的模拟器，所以多个链码并行执行并不会相互影响，模拟执行的结果也不会直接影响当前的状态数据库，生成的 `TxRwSet` 在提交交易的时候，只有验证通过以后才会记录到账本中。

## 5.5 区块索引

超级账本提供多种区块索引（`Block Index`）方式，以便能够快速找到区块。这些方式包括：

- ❑ 区块编号；
- ❑ 区块哈希；
- ❑ 交易编号；
- ❑ 同时按区块编号和交易编号；
- ❑ 区块交易编号；
- ❑ 交易验证码。

5.5.1 文件位置指针

索引的内容是文件位置指针（File Location Pointer），位置指针的结构如下所示：

```
type fileLocPointer struct {
    fileSuffixNum int
    locPointer
}

type locPointer struct {
    offset      int
    bytesLength int
}
```

文件位置指针由 3 个部分组成：

- ❑ 所在文件的编号 fileSuffixNum；
- ❑ 文件内的偏移量 offset；
- ❑ 区块占用的字节数 bytesLength。

区块查找是一个三级索引过程，查找一个区块就是先确定是哪个链，然后根据文件编号找到对应的文件，再根据文件的偏移量和占用的字节数确定区块的内容，如图 5-3 所示。

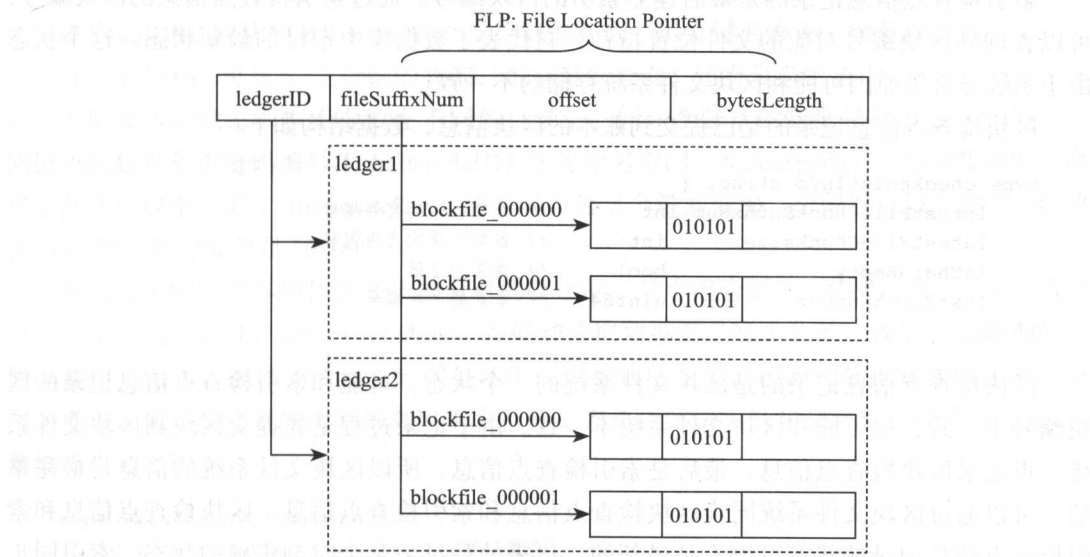


图 5-3 多链下区块的三级索引

文件位置指针序列化后保存到 LevelDB 数据库中，不同的索引方式对应的键如表 5-1 所示（“+”是逻辑符号，不是真实的键组成部分）：

构建哪些索引类型是可配置的，默认是所有类型都建立索引（排序服务也会建立索引，但只有区块编号这种索引类型）。有两种途径可以构建索引：

- ❑ 提交区块（Commit Block）；

□ 同步索引 (Sync Index)。

表 5-1 区块索引类型

索引类型	键的组成 (转换成 []byte)	索引类型	键的组成 (转换成 []byte)
区块编号	n+blockNum	同时按区块编号和交易编号	a+blockNum+txNum
区块哈希	h+blockHash	区块交易编号	b+txID
交易编号	t+txID	交易验证码	v+txID

提交区块到账本的时候会自动按照表 5-1 所示的区块类型构建索引，下面来看一下索引的同步过程。

5.5.2 索引的同步过程

索引同步只在创建区块文件管理器 (blockfileMgr) 的时候执行，验证数据库里保存的索引是否和区块文件系统里的一致。

数据库记录了两个检查点的信息：

□ 索引检查点信息 (indexCheckpointKey)；

□ 区块检查点信息 (blkMgrInfo)。

索引检查点信息记录的是最后建立索引的区块编号。通过索引检查点信息的区块编号，可以查询到区块编号对应的文件位置指针，它代表了数据库中记录的最新状态。这个状态由于系统异常等原因可能和区块文件系统存储的不一致。

区块检查点信息记录的是已提交到账本的区块信息，数据结构如下：

```
type checkpointInfo struct {
    latestFileChunkSuffixNum int    // 最新区块的文件编号
    latestFileChunksize      int    // 最新区块的文件偏移
    isChainEmpty             bool   // 是否为空链
    lastBlockNumber          uint64 // 最新的区块编号
}
```

区块检查点信息记录的是区块文件系统的状态，可能和索引检查点信息记录的区块编号不一致，也可能和区块文件系统不一致。由于记录过程是先提交区块到区块文件系统，再记录区块检查点信息，最后是索引检查点信息，所以区块文件系统的信息是最完整的。可以通过区块文件系统同步区块检查点信息和索引检查点信息。区块检查点信息和索引检查点信息记录的状态可能不是最新的，代表的是过去某个时刻正确的状态，索引同步可以从检查点记录的状态开始，更新为区块文件系统的最新状态，减少同步时间。

区块文件管理器在创建的时候会从数据库中获取区块检查点信息，并且需要和区块文件系统对比检查其是否为最新的状态。检查过程是基于区块检查点的文件编号和文件内由偏移构建的一个区块文件流 (blockfileStream)，从这个偏移开始验证该文件是否还有区块数据。如果确实还有区块，找到该文件实际存储的最大区块编号，更新区块检查点信息。这里用的是区块文件流，它只会检查单个文件。这可能会存在一个特殊情况，就是上一个区

块刚好写满这个文件，达到了上限 `maxBlockfileSize`，这个区块从下一个文件开始写，写完区块数据以后，更新区块检查点信息的时候出现异常，这时会导致区块检查点信息记录和区块文件系统的永久性不一致。

经过上面的检查后，区块检查点信息就和区块文件系统完全一致了。根据区块检查点信息，构建区块流，从索引检查点信息记录的区块文件和偏移开始，到区块检查点记录的最新区块的文件编号为止，全部重新构建索引，并更新索引检查点信息。

## 5.6 状态数据

状态数据 (State Database) 记录的是交易执行的结果，最新的状态代表了通道 (Channel) 上所有键的最新值，所以又称为“世界状态”。链码调用根据当前状态数据执行交易。为了提高链码执行的效率，所有键的最新值都存储在状态数据库中。状态数据库只是区块链交易日志中的索引视图，因此可以随时根据区块链重新生成。状态数据库在 Peer 节点启动时自动恢复，重新构建完成后才接受新的交易。

对于状态数据库本身插件化的设计，目前支持 LevelDB 和 CouchDB。LevelDB 和 CouchDB 都支持基本的链码操作，比如获取和设置键值，基于键进行查询等。

□ LevelDB (默认的 KV 数据库)：支持键的查询、组合键的查询、键范围查询。

□ CouchDB (可选)：支持键的查询、组合键的查询，还有复杂的查询。

不同账本的状态数据库存放在不同的目录下，同一个账本的数据是存放在一起的，不同链码的数据是按链码编号 (chaincodeID) 作为命名空间 (Namespace) 来划分数据的。命名空间在生成组合键 (compositeKey) 的时候作为组合键的前缀，分隔符可以自定义 (修改源码)，默认的分隔符是 "0x00"。

状态数据库的基本操作是基于键值对的管理的。某个组合键  $k$ ，在指定版本 `ver` 的值可以用一个三元组：( $k$ , `ver`, `val`) 表示，存储和读取数据都是版本化的。读取状态数据的时候不能指定版本，读取到的状态数据是某个时刻最新的版本，返回的数据包含版本和数据两个部分，版本的实现用交易的高度来表示的，它是由区块编号和交易编号组成的二元组。数据结构定义如下：

```
type VersionedValue struct {
    Value    []byte           // 数据
    Version  *version.Height // 版本
}

type Height struct {
    BlockNum uint64 // 区块编号
    TxNum    uint64 // 交易编号
}
```

基本区块数据的读取也是通过键来查询的，有 3 种方式：查询单个键的数据、查询多



个键的数据、查询一个范围内的数据。如果采用的是 CouchDB 方式，还可支持某些字段的条件查询，详细的内容下面的章节会有介绍。

节点验证完数据后会披露并进行更新。这可以同时包含不同链码上的数据，同时用账本编号（chaincodeID）作为命名空间分割。相同链码的数据是由不同的键值对组成的字典（map）。批量更新的数据各自有不同的版本，在更新同一批数据的同时会更新世界状态交易的高度。

状态数据库支持如下的功能。

1) 根据命名空间和键获取状态数据。

□ 获取单个键的数据：GetState

□ 获取多个键的数据：GetStateMultipleKeys

□ 获取一个范围内的查询数据：GetStateRangeScanIterator

2) 根据条件查询获取数据：ExecuteQuery。

3) 更新状态数据：ApplyUpdates，可批量更新。

4) 获取最新交易高度：GetLatestSavePoint。

5) 数据库操作。

□ 打开数据库：Open

□ 关闭数据库：Close

### 5.6.1 LevelDB

LevelDB 是默认的状态数据库。LevelDB 是采用 C++ 编写的一种高性能嵌入式数据库，没有独立的数据库进程，占用资源少，速度快。它有如下一些特点。

1) 键和值可以是任意的字节数组。

2) 数据是按键排序后存储的。

3) 可以自定义排序方法。

4) 基本的操作是基于键的：

□ Put(key, value);

□ Get(key);

□ Delete(key)。

5) 支持批量修改的原子操作。

6) 支持创建快照。

7) 支持对数据前向和后向的迭代操作。

8) 数据采用 Snappy 压缩。

超级账本基于 <https://github.com/syndtr/goleveldb> 实现对 LevelDB 数据库的操作。状态值的存储和获取都是基本的键值操作，键是包含了链码编号（chaincodeID）的组合键，值是包含了版本信息和状态值的序列化结果。由于 LevelDB 不支持复杂的查询，所以使用

LevelDB 作为状态数据库也不支持条件查询接口：ExecuteQuery。交易高度是一个区块编号和交易编号组成的二元组，存储也是按照键值对的方式进行的，键为“0x00”；值是二元组序列化后的结果，GetLatestSavePoint 是反序列化后得到的交易高度。具体的序列化和反序列化方法见后面的实现部分。

### 5.6.2 CouchDB

另外一个可选的数据库是 CouchDB。CouchDB 是一种文档型数据库，提供 RESTful 的 API 操作数据库文档。CouchDB 中的文档是无模式的（Schemaless），并不要求文档具有某种特定的结构。CouchDB 支持原生的 JSON 和字节数组的操作，基于 JSON 的操作，可以支持复杂的查询。如果存储的数据是字节数组，也支持基本的键值对操作。存储在 CouchDB 中的数据 CouchDoc 包含 JSONValue 和附件两个部分，如下所示。

```
type CouchDoc struct {
    JSONValue []byte
    Attachments []Attachment
}
```

```
type Attachment struct {
    Name          string
    ContentType    string
    Length        uint64
    AttachmentBytes []byte
}
```

其中 JSONValue：也和存储的类型有关系，它最终会转换成一个 JSON 的结构。JSONValue 结构序列化后的内容如下。

```
{
  "version": "$BlockNum:$TxNum",
  "chaincodeid": "$chaincodeID",
  "data": "$rawJSON"
}
```

在超级账本中，如果存储的类型是 JSON，且 JSONValue 的 data 字段是状态值经过 JSON 序列化后的内容，则 CouchDoc 中的 Attachments 为空；如果存储的类型是字节数组，则 JSONValue 只保存版本信息，data 字段为空，状态值放在 Attachments 的 AttachmentBytes 中，Attachments 的 Name 为“valueBytes”，ContentType 为“application/octet-stream”。在获取时根据 data 字段是否为空可以判断出存储的状态值类型，最后得到存储的版本和状态值。

条件查询是基于 LevelDB 的状态数据库所没有的功能进行的。查询前会对查询条件进行转换，增加“data.”前缀和查询记录限制等，查询结果还会默认增加“\_id”“version”“chaincodeid”。比如原始的查询条件为：

```

{
  "selector": {
    "owner": {
      "$eq": "tom"
    }
  },
  "fields": [
    "owner",
    "asset_name",
    "color",
    "size"
  ],
  "sort": [
    "size",
    "color"
  ]
}

```

转换后的查询条件为:

```

{
  "selector": {
    "$and": [
      {
        "chaincodeid": "marble"
      },
      {
        "data.owner": {
          "$eq": "tom"
        }
      }
    ]
  },
  "fields": [
    "data.owner",
    "data.asset_name",
    "data.color",
    "data.size",
    "_id",
    "version",
    "chaincodeid"
  ],
  "sort": [
    "data.size",
    "data.color"
  ],
}

```

```

    "limit": 10,
    "skip": 0
}

```

其中查询记录限制 limit 可以在 ledger.state.queryLimit 中设置，默认值为 1000。

交易高度也是采用键值存储的，当键为 statedb\_savepoint，值为 couchSavepointData 时 JSON 的结构如下所示。

```

type couchSavepointData struct {
    BlockNum  uint64 `json:"BlockNum"`
    TxNum     uint64 `json:"TxNum"`
    UpdateSeq string `json:"UpdateSeq"`
}

```

### 5.6.3 基于状态数据的区块验证

区块数据提交到账本前，会基于状态数据验证区块数据是否有效。区块验证之前会从区块中解析出有效载荷（详细的消息结构参见后面的章节），根据不同的类型分别进行验证，目前的区块类型有：

□ 背书交易区块

□ 配置交易区块

配置交易区块的验证目前暂时没有实现，验证结果都为成功。我们来看看背书交易区块的验证过程，交易区块的验证主要是读写集的验证。从 ChaincodeAction 中解析出读写集 TxRwSet，验证读取的版本是否和状态数据库里的版本一致。如果读写集中还有范围查询，也会验证范围查询中的每个记录是否和状态数据库中的版本完全一致。验证通过的交易，其读写集会添加到 UpdateBatch 中，同一个区块的交易还会验证是否会读取 UpdateBatch 中的记录，因为 UpdateBatch 中的记录是当前区块更新的数据，读取还没有提交到账本中的数据，这会导致和模拟执行时读取的数据版本不一致，所以在这种情况下验证会失败。

区块提交到账本之后，要从区块数据中恢复状态数据和历史数据，就不会再对读写集进行版本检查。

数字签名信封，没有验证签名。

交易验证过程实现了一个验证字节图，每个交易占用一个字节，并标识其状态，定义如下：

```

type TxValidationFlags []uint8

```

验证后的字节图会存放在无数据中。验证成功的背书交易区块会生成读写集。

#### 1. 基于版本的验证

基于状态数据版本的验证过程比较简单，对每一个状态的 kvRead.Key，比较状态数据里保存的版本是否和交易记录里面读取的版本一致，若完全一致就验证通过，表示在模拟

执行之后这个 `kvRead.Key` 的值没有被修改过。

## 2. 基于范围查询的验证

基于范围查询的验证方法是：比较“范围查询的结果”是否和在最新状态数据基础上更新本次交易数据后的“模拟状态数据”一致，有两种方法可以进行比较：

□ 基于默克尔树计算哈希值的比较；

□ 基于范围查询结果的查询。

如果范围查询的结果已经包含了默克尔树哈希计算结果，范围查询就采用基于默克尔树计算哈希值的比较方法。比较的过程是逐个计算模拟状态数据的默克尔树哈希，在计算过程中判断是否和范围查询的默克尔树哈希值一致，若过程中出现不一致就退出。默克尔树的计算过程对新增元素是友好的，可以在已经计算过的结果基础上迭代计算出最后的结果，不需要整个数组一起重新计算。

我们再来看看基于范围查询结果的查询比较过程，比较的方法就是遍历每个元素，比较两个迭代器元素的键值对是否完全一致，包括元素的个数。

## 5.7 历史数据

历史数据（History Database）记录了每个状态数据的历史信息，历史信息是保存在 LevelDB 数据库中的。每个历史信息用一个四元组（namespace、writeKey、blockNo、tranNo）来表示，其中：

□ namespace：实际代表的是不同的 chaincodeID，从这里也可以看出，不同 chaincode 的数据是逻辑隔离的；

□ writeKey：要写入数据的键；

□ blockNo：要写入数据所在的区块编号；

□ tranNo：要写入数据所在区块内的交易序号，从 0 开始。

历史信息记录最细的粒度就是交易，如果在一个交易中多次对同一个 writeKey 更新数据，则会以第一次数据为准，历史信息实际存储的信息是固定的空字节数组 `[]byte{}`。更新区块信息的时候，会同步更新检查点信息，保存的内容是最新的区块高度和最大的交易序号，检查点信息用来判断历史信息的状态是否是最新的。

## 5.8 数据恢复

区块的提交过程分为 3 个步骤：

□ 先保存区块到文件存储的账本数据中；

□ 然后更新状态数据；

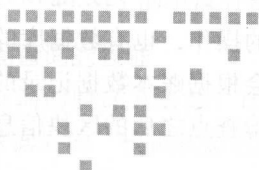
□ 最后更新历史信息数据。

这3个步骤是顺序执行的，在这个过程中有文件的操作，也有数据库的操作，所以在任何一个步骤都可能出现错误或者中断。恢复的过程会根据账本数据记录的区块信息和状态数据、历史信息数据的检查点进行比较，重新提交检查点之后的区块信息，保持账本数据的一致性。

## 5.9 本章小结

本章介绍了 Hyperledger Fabric 1.0 的数据存储，包括账本数据 (Ledger)、区块索引 (Index)、状态数据 (state Database)、历史数据 (History Database) 等存储结构。目前的数据存储存在较大的优化空间，账本数据结构的设计带来的开销很大，最新的版本并没有实现账本裁剪 (Ledger Prune) 功能，不能进行归档，也不能删除无效交易，所以会导致存储空间持续增长。





## 集成共识机制的排序服务

本章会介绍共识的一些基本概念和 Hyperledger Fabric 1.0 中的共识机制，及其可插拔的架构设计。

### 6.1 概述

在区块链系统中，共识（Consensus）是多个参与方对一个交易是否提交到账本以及提交的顺序达成一致的过程。由于是多个节点参与的分布式系统，所以网络传输可能存在延时。共识一致并不代表在所有时刻都有完全相同的结果，是经过一段收敛时间后，网络中的多数节点对同一个交易执行相同的记账操作。在共识的过程中可能存在一些节点无响应或者响应延迟的情况，也可能存在一些参与方恶意提交请求或者篡改请求内容的情况。根据错误类型的不同，共识算法可以满足两种范围的容错。

1) 崩溃故障容错（Crash Fault-Tolerance, CFT）：在区块链网络中存在网络延时或者故障的情况下，共识机制能够确保有效的交易达成一致。

2) 拜占庭容错（Byzantine Fault-Tolerance, BFT）：在区块链网络中存在部分恶意节点提交或者篡改请求的情况下，共识机制能够确保有效的交易达成一致。

在共识过程中网络节点要确保交易有序且交易区块有效，需要提供以下核心功能。

1) 交易有效：能够根据背书及共识策略确保区块中所有交易有效。

2) 交易有序：能够确保所有节点提交和执行交易顺序的一致性，这才能保证执行结果的一致性和最终全局状态的一致性。

3) 交易验证：能够利用智能合约的接口，验证交易的有效性和提交顺序。

根据第 3 章的交易流程，我们看到在 Hyperledger Fabric 1.0 中，一个交易从提交到最终记账会经历多个阶段，每个阶段都需要多节点的参与，共识是在分阶段共识基础上的全过程共识，对于任何一个阶段的共识失败，最终结果都是共识失败。

6.1.1 共识算法的类型

共识算法必须具备两个特性以保证节点之间数据的一致性。

1) 安全性 (Safety)：它指每个节点保证相同的输入序列，并在每个节点上产生相同的输出结果。当节点接收到相同顺序的交易时，每个节点将发生同样的状态改变。算法的执行结果必须与单节点系统依次执行每个交易的结果一致。

2) 存活性 (Liveness)：指在没有通信故障的情况下，每个非故障节点最终都能接收到提交的所有交易。

节点一致性在账本上的体现是：

1) 区块的确定性：在不同节点上相同区块号的区块内容完全一致；

2) 区块的完整性：在不同节点上有相同的区块数，且按照顺序形成相同的区块链。

共识算法大体可以分为两种类型，基于彩票中奖的算法 (Lottery-based Algorithm) 和基于投票计数的算法 (Voting-based Algorithm)。

基于彩票中奖的算法是一个形象化的说法，加入到区块链网络中的节点都可以生成区块，广播给网络中的其他节点以进行确认，通过竞争获得最终的记账权。实际上，区块是否获取其他节点的确认是有一定概率的，这与买彩票中彩一样。这种算法是一种先记账再共识的算法，优势是其可以很容易地扩展到大量的节点上，缺点是区块记账是一种大概率的确认真算法，区块确认的时间较长。常见的一些算法，如消逝时间量证明算法 (Proof of Elapsed Time, PoET) 和工作量证明算法 (Proof of Work, PoW) 都属于这一类算法。

基于投票计数的算法是节点参与区块的投票，网络中的节点确认区块以后再记账的过程。这种算法中区块的记账是确定性的，已经记账的区块都是有效的。这是一种绝对一致的机制，确定的结果才能在某些对结果要求比较高的场合使用，比如金融场景，已经完成的转账是不能随意地撤回的。这种算法的缺点是需要更长的时间达成共识，不能满足某些需要高吞吐量、低延迟的场景要求。这就需要不同的应用场景在可扩展性、确定性和速度等方面进行一个权衡。冗余拜占庭容错算法 (Redundant Byzantine Fault Tolerance, RBFT) 和 Paxos 都属于这一类型的算法。

表 6-1 所示为这两种算法的比较。

表 6-1 共识算法类型的比较

	基于彩票中奖的算法	基于投票计数的算法
节点可扩展性	好	一般
交易确定性	大概率一致的共识机制	绝对一致的共识机制
交易完成速度	快	慢

(续)

	基于彩票中奖的算法	基于投票计数的算法
共识网络流量	少	多
共识算法特点	先记账再共识	先共识再记账
典型算法示例	PoET、PoW	RBFT、Paxos

超级账本应用场景的目标是企业级的区块链平台，网络节点一般是需要授权加入的，所以超级账本是不支持标准的 PoW 算法的。

6.1.2 Hyperledger Fabric 1.0 的共识机制

超级账本中是把共识分为 3 个阶段，如图 6-1 所示。

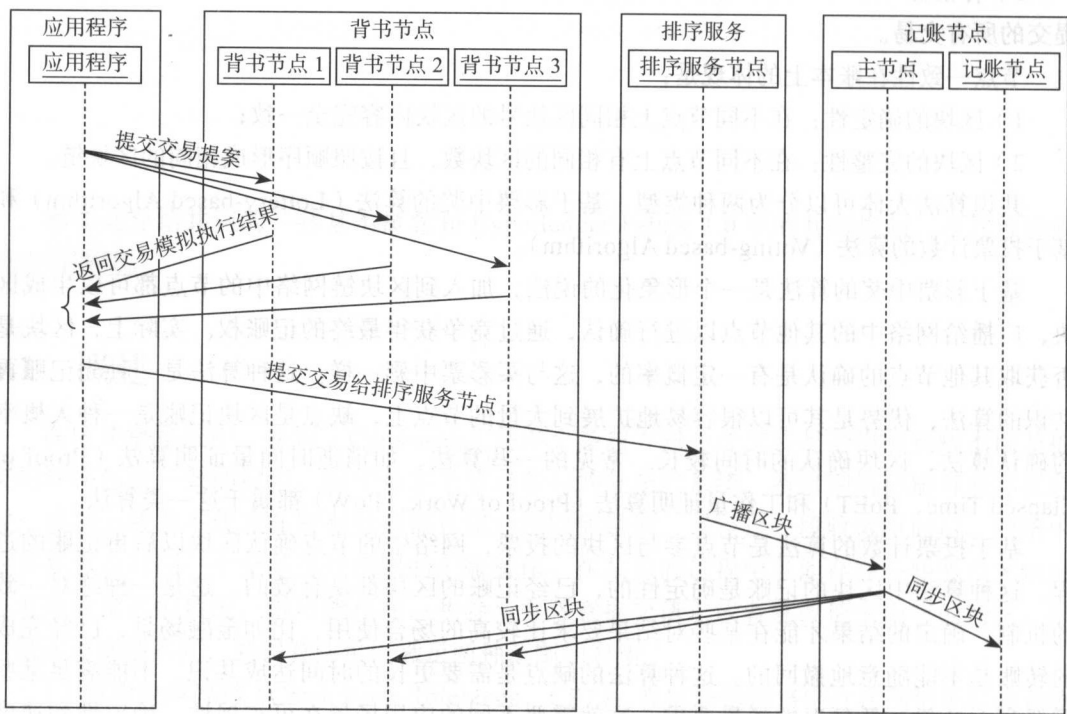


图 6-1 超级账本的共识机制

(1) 交易背书

应用程序根据背书策略的要求选择背书节点，给这些节点发送需要执行的交易提案 (Proposal)。背书节点调用链码 (Chaincode) 执行这些交易提案，交易过程是执行是模拟执行的，并不真正提交数据到账本中。执行完成以后调用交易背书系统链码 ESCC 对模拟执行结果进行签名背书。

(2) 交易排序

排序阶段接受已经签名背书的交易，确定交易的顺序和数量，将排好序的交易打包到

区块中，广播给 Peer 节点进行验证。通常，从效率方面考虑，排序服务不会输出单个交易作为一个区块，而是把多个交易打包成一个区块。

### (3) 交易验证

Peer 节点验证接收到区块里包含的交易的有效性，包括背书策略验证及双花检测 (double-spending)。可以将验证错误分为两大类：语法错误和逻辑错误。语法错误包括无效输入、未验证的签名以及重复的交易（双花攻击），重复的交易应该丢弃。第二类错误比较复杂，比如会导致双花或者 MVCC 失败的交易，这需要定义策略来决定程序是继续执行还是终止。策略还可以定义是否需要记录日志以便对这类交易进行审计。交易验证依赖链码的业务逻辑，目前默认的交易验证系统链码 VSCC 只支持背书策略的验证，详细的过程后面的章节会有介绍。

在超级账本中这 3 个阶段的设计均是可插拔的，应用程序可根据自身需求实现和选择不同的交易背书、交易排序及交易验证模块。

我们先来看下交易背书和交易验证的可插拔设计。交易背书和交易验证由内置的系统链码 (System Chaincode) 来实现，这两个系统链码都是可替代的。可以把交易背书和交易验证的功能逻辑实现成链码 (Chaincode) 并提交到链上，在提交交易提案的时候指定新的交易背书和交易验证链码。前面的章节已经详细介绍过 peer chaincode 命令，里面的“-E”和“-V”参数就是指定新的交易背书和交易验证系统链码的。

排序服务的实现也是可插拔的，它采用的是异步事件的方式，提供了两个基本的接口：

1) broadcast (blob): 客户端调用 broadcast 接口在通道上广播 blob 消息。

2) deliver (seqno, prevhash, blob): 排序服务调用 deliver 给客户端发送 blob 消息，包含序号 seqno (无符号整数) 和上一个消息的哈希 (prevhash)，它是排序服务的事件输出接口。

目前，正式发布的版本只支持 Apache Kafka 的排序服务。排序服务接口会加入基于 BFT 协议的算法，目前正在开发中的算法有 BFT Smart、简化拜占庭容错算法 (SBFT)、蜜罐拜占庭容错算法 (Honey Badger of BFT) 等。在本章的后面部分我们会详细介绍排序服务的接口，大家可以实现一个自定义的排序服务。

## 6.2 实现数据隔离的多通道

排序服务给客户端和 Peer 节点提供了一个共享通信通道 (Communication Channel)，用来实现交易的广播服务。客户端连接到通道 (Channel) 上，在通道上广播的消息会最终发送给通道内所有的 Peer 节点。通道支持消息的原子广播 (Atomic Broadcast)，通道给所有相连的 Peer 节点输出相同的消息，并且有相同的逻辑顺序。这种原子通信保证也叫全序广播 (Total-order Broadcast)。

排序服务支持多通道 (Multi-channel)，类似 Kafka 消息系统的主题 (Topics)。客户端

连接到一个指定的通道上，就可以发送或者获取消息了。通道是相互隔离的，客户端连接到一个通道是不知道其他通道的存在的，但是客户端可以连接到多个通道。为简单起见，在本章后面的部分，除非明确提到的其他情况，否则我们都假设排序服务是由单个通道和主题组成的。

图 6-2 所示为一个多通道的示例，基于 SDK 开发的应用程序或者命令行程序通过 gRPC 通道给排序服务提交背书节点以模拟执行后的交易，排序服务会根据交易信息里提案的请求头确定通道信息，添加到对应的队列中进行排序，生成区块后广播给加入了这个通道的节点。比如节点 1 会收到通道 1 和通道 N 的区块信息，节点 2 会收到通道 1、通道 2 和通道 N 的区块信息，节点 N 会收到通道 2 和通道 N 的区块信息。没有加入通道中的节点是接收不到这个通道的区块信息的，比如节点 1 收不到通道 2 的区块信息，节点 N 收不到通道 1 的区块信息。

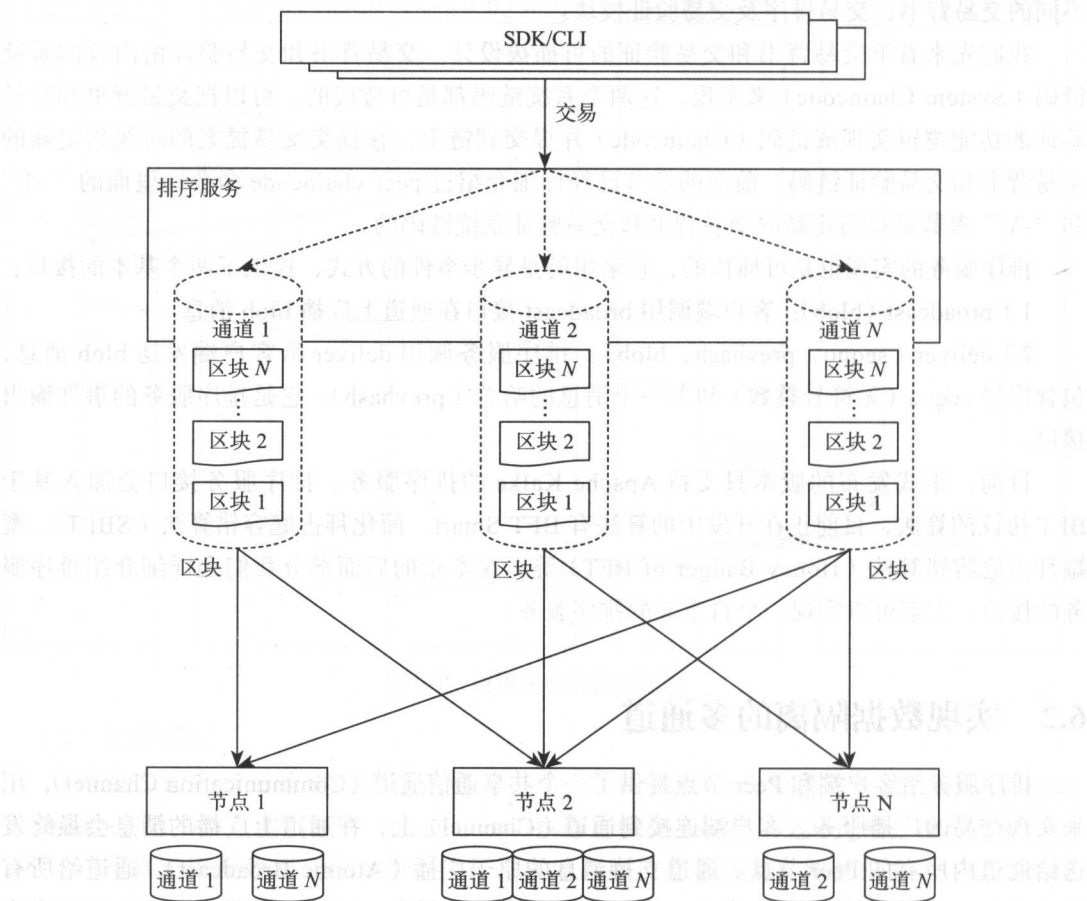


图 6-2 多通道示例

每个通道在节点上都有一个关联的账本，节点 2 在 3 个通道上，本地会对应有 3 个账

本数据，账本存储结构参考上一章的介绍。在区块链网络的所有节点上都存在的账本称为系统账本，反之称为子账本。从数据隔离角度看，两个通道上的节点完全一样是没有意义的，有多个系统账本也是没有意义的。由于排序服务能接收到所有链的交易信息，所以数据隔离是对节点来说的，并不针对排序服务节点（Ordering Service Nodes, OSN）。如果业务上希望交易信息的内容对排序服务也保密，可以对交易信息进行哈希或者加密，避免传输明文信息。

### 6.2.1 排序服务的初始化

排序服务是由多个排序节点组成的，在每个排序节点启动的时候需要有一个创世区块（Genesis Block）。创世区块包含的信息有：

- ❑ 排序节点信息及其 MSP 信息（管理员证书、根证书和 TLS 根证书）；
- ❑ 组织信息及其 MSP 信息（管理员证书、根证书和 TLS 根证书）；
- ❑ 共识算法类型；
- ❑ 区块配置信息；
- ❑ 访问控制策略。

创世区块是通过 configtxgen 工具生成的，还有一个工具 configtxlator 可以实现区块和 JSON 格式之间的相互转换，下面是一个创世区块转换成 JSON 后的格式，包含的信息概要如下所示：

```
{
  "data": {
    "data": [
      {
        "payload": {
          "data": {
            "config": {
              "channel_group": {
                "groups": {
                  "Consortiums": {
                    "groups": {
                      "SampleConsortium": {
                        "groups": {
                          "Org1MSP": {...
                        },
                        "Org2MSP": {...
                      }
                    },
                    "mod_policy": "/Channel/
                    Orderer/Admins",
                    "values": {...
                  }
                }
              }
            }
          }
        }
      }
    ]
  }
}
```



```

    },
    "mod_policy": "/Channel/Orderer/
Admins",
    "policies": {...
    }
  },
  "Orderer": {
    "groups": {
      "OrdererOrg": {...
    }
  },
  "mod_policy": "Admins",
  "policies": {...
  },
  "values": {
    "BatchSize": {
      "mod_policy": "Admins",
      "value": {
        "absolute_max_bytes":
          102760448,
        "max_message_count": 10,
        "preferred_max_bytes":
          524288
      }
    },
    "BatchTimeout": {
      "mod_policy": "Admins",
      "value": {
        "timeout": "2s"
      }
    },
    "ChannelRestrictions": {
      "mod_policy": "Admins"
    },
    "ConsensusType": {
      "mod_policy": "Admins",
      "value": {
        "type": "solo"
      }
    }
  }
},
"mod_policy": "Admins",
"policies": {...
},
"values": {
  "BlockDataHashingStructure": {
    "mod_policy": "Admins",
    "value": {

```

```

        "width": 4294967295
      }, {
        "HashingAlgorithm": {
          "mod_policy": "Admins",
          "value": {
            "name": "SHA256"
          }
        },
        "OrdererAddresses": {
          "mod_policy": "/Channel/Orderer/
Admins",
          "value": {
            "addresses": [
              "orderer.example.com:7050"
            ]
          }
        }
      }
    ],
    "header": {
      "data_hash": "etbznpwMod/5MUv9j3U19fo8fwj6WYd+0PFWFx2c2X8="
    },
    "metadata": {
      "config_update": {
        "channel_id": "mychannel",

```

其中，…表示信息省略了。

## 6.2.2 通道的创建

应用程序可以通过 SDK 或者命令行向排序服务发起创建通道的请求，提交的内容是通道配置交易（Channel Configuration Transaction）。通道配置交易需要由工具 configtxgen 生成，详细步骤在第 2 章已经详细介绍过。以下是一个通道配置交易通过工具 configtxlator 转换成 JSON 格式的示例：

```

{
  "payload": {
    "data": {
      "config_update": {
        "channel_id": "mychannel",

```

```

"read_set": {
  "groups": {
    "Application": {
      "groups": {
        "Org1MSP": {},
        "Org2MSP": {}
      }
    }
  },
  "values": {
    "Consortium": {
      "value": {
        "name": "SampleConsortium"
      }
    }
  }
},
"write_set": {
  "groups": {
    "Application": {
      "groups": {
        "Org1MSP": {},
        "Org2MSP": {}
      },
      "mod_policy": "Admins",
      "policies": {
        "Admins": {
          "mod_policy": "Admins",
          "policy": {
            "type": 3,
            "value": {
              "rule": "MAJORITY",
              "sub_policy": "Admins"
            }
          }
        }
      }
    },
    "Readers": {
      "mod_policy": "Admins",
      "policy": {
        "type": 3,
        "value": {
          "sub_policy": "Readers"
        }
      }
    },
    "Writers": {
      "mod_policy": "Admins",
      "policy": {
        "type": 3,
        "value": {

```

```

        "sub_policy": "Writers"
    }
}
},
"version": "1"
}
},
"values": {
    "Consortium": {
        "value": {
            "name": "SampleConsortium"
        }
    }
}
},
"header": {
    "channel_header": {
        "channel_id": "mychannel",
        "timestamp": "2017-08-22T18:53:29.000Z",
        "tx_id": "e3b0c44298fc1c149afb4c8996fb92427ae41e4649b934ca495991b7852b855",
        "type": 2
    }
}
}
}
}

```

排序服务接收到创建通道的请求，会检查是否是配置交易。检查的方法是查看通道头 ChannelHeader 的类型是否为 HeaderType\_CONFIG\_UPDATE (新建通道和更新通道配置的类型都是这个类型)，然后排序服务节点重新生成一个配置交易，修改交易类型为 HeaderType\_CONFIG，内容除了包含 write\_set 里的 Application，还会包含系统链里的 Orderer 和 Consortium，新生成的交易会利用接收消息的排序服务节点的私钥重新进行签名，然后添加到系统链的交易消息队列中进行处理。

每个链（包括系统链）有一个协程专门处理消息队列中的消息，按消息队列的顺序进行排序。系统链增加了 newSystemChainFilter 的消息过滤器，在排序的过程中会检查系统链中的消息是否满足过滤器的条件，检查的结果可能是转发、拒绝或者接收，接收消息的同时返回系统链的提交器 systemChainCommitter。交易切割形成区块后，在区块写入系统账本之前会调用提交器创建一个新的链，在新的链上创建一个包含配置交易的创世区块。新的链会在多账本管理器 multiLedger 处注册，当接收到这个链上的交易请求时，排序服务节点会分发给这个链来处理。这样，排序服务节点就有了多链的处理逻辑，给不同链的交易划分了不同的通道，实现了数据的隔离。

应用程序发起了创建通道的请求，创建成功后返回这个链的创世区块。新链的创世区块如下：

```
{
  "data": {
    "data": [
      {
        "payload": {
          "data": {
            "config": {
              "channel_group": {
                "groups": {
                  "Application": {
                    "groups": {
                      "Org1MSP": {...},
                      "Org2MSP": {...}
                    },
                    "mod_policy": "Admins",
                    "policies": {...},
                    "version": "1"
                  },
                  "Orderer": {...}
                },
                "policies": {...},
                "values": {...}
              },
              "sequence": "1"
            },
            "last_update": {
              "payload": {
                "data": {
                  "config_update": {
                    "channel_id": "mychannel",
                    "read_set": {...},
                    "write_set": {...}
                  },
                  "signatures": [...]
                },
                "header": {...}
              },
              "signature": "MEUCIQDoU1zwztKkRhp5f0X2aWef7xQ73P7rquMrGGx7EcRNNQIge2inBpxbGHkQ83lyHoMDa7eciNpmX3+QwCLbzOgkUpU="
            }
          }
        }
      }
    ]
  }
}
```

```

    }
  },
  "header": {
    "channel_header": {
      "channel_id": "mychannel",
      "timestamp": "2017-08-31T09:24:31.000Z",
      "type": 1
    },
    "signature_header": {...
  }
},
"signature": "MEQCIA/fOYvcOou5pkHtjtiG3eV4f67z4D/
Myusxu3qAHAdAAiB0j/ekiakdHkbOBM9uuR2bzYjVBrKuoK1UaJ1ZuWkm8Q=="
}
}
},
"header": {...
},
"metadata": {...
}
}
}

```

新链的创世区块包含了通道配置交易的内容，扩展了从系统链上保存的组织信息、排序节点服务信息等，这样节点可以根据这个创世区块确定新链的标识、排序服务节点地址等，而不用访问排序服务的系统链。

### 6.2.3 通道的更新

通道的配置是可以更新的，包括通道组织、组织 MSP、访问控制策略、排序服务配置等。通道更新需要线下确定好需要修改的配置项，离线修改后通过 SDK 或者 CLI 发送给排序服务节点，详细的过程如图 6-3 所示。

从图 6-3 可以看到，通道配置的修改同样需要利用 configtxlator 服务，详细的使用方法参考附录 B 的内容。

发送给排序服务的交易请求类型为 HeaderType\_CONFIG\_UPDATE，排序服务节点接收到配置更新交易请求后会重新创建一个类型为 HeaderType\_CONFIG 的信封，添加到被修改通道的交易队列里。每个链都有一些过滤器规则，若检测到是类型是 HeaderType\_CONFIG 就提交给 configManager 来处理，Apply 函数处理过程如图 6-4 所示。

利用排序服务生成的配置区块，也会通过排序服务建立好 gRPC 连接的主节点 (orgLeader) 广播给通道内的其他节点。记账节点接收到配置区块以后，账本提交器 (LedgerCommitter) 在提交到账本之前会检查 ChannelHeader 的类型，如果是 HeaderType\_CONFIG 就确认是配置区块，从区块里获取链编号，更新本地节点这个链的配置区块为最新收到的区块。



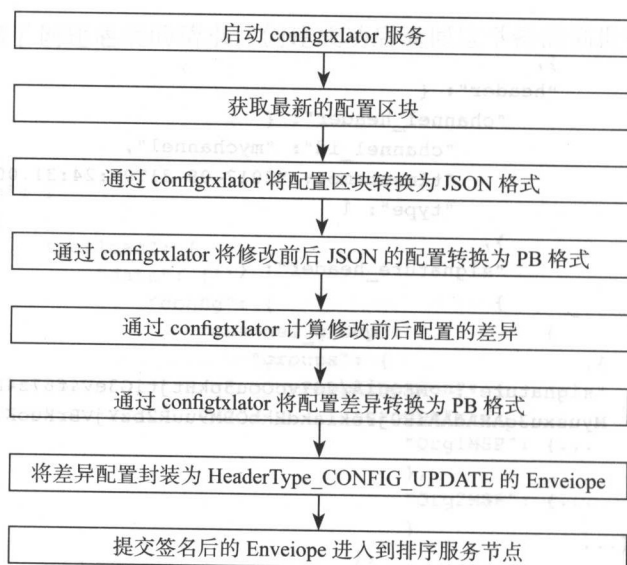


图 6-3 利用 configtxlator 构造通道更新请求

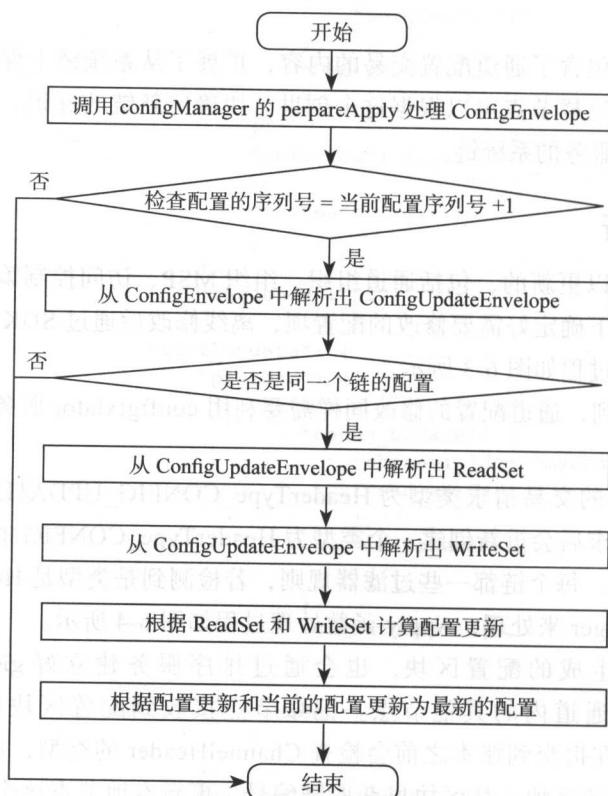


图 6-4 排序服务节点处理通道更新请求

### 6.2.4 通道的加入

应用程序通过 SDK 或者命令行给节点发送包含通道创世区块的 JoinChain 请求, 请求的类型是 HeaderType\_CONFIG。节点加入链的 JoinChain 请求是由系统链码 CSCC 处理的。节点会校验创世区块的合法性(包括内容完整性), 比如是否包含应用相关的配置项, 还会对提交者的身份进行认证和权限检查。配置交易是需要应用程序用提交者的私钥进行签名的, 身份认证需要确认请求头是否包含有效的提交者身份信息, 再验证配置交易请求是否是该提交者签名的。然后是权限检查, 包括两个方面。

1) 是否有权向节点提交请求: 即检查提交者的 MSP 是否与本地 MSP 相同。

2) 是否满足加入通道请求的策略: 提交加入通道请求是需要有管理员权限的, 管理员的证书配置在节点的 \$CORE\_PEER\_MSPCONFIGPATH/admincerts 目录下。

权限检查会在节点本地创建以请求头里 channel\_header 的 channel\_id 为标识的链, 节点的本地配置是由主节点(orgLeader)或者选举为主节点后主动和创世区块里设置的排序服务节点建立连接, 接收排序服务广播的新区块, 再在组织内部节点同步; 节点若不是主节点就从组织内部其他节点上同步, 详细的过程参见上一章的相关内容。

### 6.2.5 通道的查询

通道信息是节点本地维护的, 有一个值为 chain 的映射表(list map[string]\*chain), 在节点启动或者有加入通道等操作的时候更新这个映射表。映射表的键就是通道名称, 遍历就能返回节点加入的所有通道。这个映射表只能通过系统链码 CSCC 查询, 命令行的方式是:

```
peer channel list
```

应用程序也可以通过 SDK 发送请求, 比如 fabric-sdk-go 提供的接口 QueryChannels(peer Peer)(\*pb.ChannelQueryResponse, error) 可以查询指定节点加入的通道。命令行和 SDK 的方式都是发送相同的请求, 即构造一个调用 CSCC 的 GetChannels 调用请求。查询通道信息是需要权限的, 权限的策略配置是在配置区块里的, 一般的读取权限是只要成为这个组织的成员就可以了。通过权限检查返回的是由本地节点维护的已加入通道的名称数组。

通道的配置也是可以查询的, 比如获取排序服务节点。节点内部查询配置区块是通过调用系统链码 CSCC 的 GetConfigBlock 获取的, 同样会检查是否有权限查询, 通过检查会返回由本地维护的最新配置区块。

应用程序和命令行是怎么从账本里找到最新的配置区块呢? 实际上, 每个区块的元数据里都有最新配置区块的索引, 所以需要先给排序服务发送 SeekPosition\_Newest 请求以获取最新的区块, 从区块里解析出 BlockMetadataIndex\_LAST\_CONFIG 以获取配置区块高度, 根据区块高度获取指定的配置区块。需要说明的是, 最新的配置区块都是全量的, 如

果通道的配置信息更新过，更新的内容都会更新到最新的配置区块里，获取最新的配置区块就能查询最新的配置信息了。

命令行获取通道配置的命令：

```
peer channel fetch config config_block.pb -o $OSN_IP:$OSN_PORT -c $CHANNEL_ID
```

其中：OSN\_IP 和 OSN\_PORT 是排序服务节点的地址和端口，CHANNEL\_ID 是通道编号。config\_block.pb 是配置区块文件，如果不提供这个参数，则默认生成的配置区块文件是 \$CHANNEL\_ID.block。

通道上的交易处理和智能合约参考上一章的相关部分。截止到目前的版本，通道创建以后还不能终止或者删除，也没有提供功能让节点退出一个通道。

## 6.3 可插拔的排序服务

排序服务是可插拔的，代码里提供了几种实现：

❑ 基于单进程 (Solo) 的排序服务；

❑ 基于 Kafka 的排序服务。

目前的版本没有基于 \*BFT 的排序服务。

### 6.3.1 排序服务接口

排序服务的业务需求可以实现不同的逻辑，Hyperledger Fabric 1.0 已经预留了一些接口，需要修改地方如下所示：

❑ 创建链的接口；

❑ 链消息处理的接口；

❑ 增加新的排序服务支持。

#### 1. 创建链的接口

创建链的接口定义如下：

```
// Consenter 定义了后台的排序机制
type Consenter interface {
    // 创建并返回一个对 Chain 的引用，用于提供资源
    // 每个进程会被指定的 chain 调用一次。通常情况，发生错误不可恢复，并会导致系统关闭，有关的
    // 详细信息，请参考 Chain 的描述
    // 第二个参数是一个指针，指向该 Chain 中账本最后一个提交块 ORDERER 的存储元数据。由于
    // genesis 块中没有存储元数据字段定义，所以在新的 Chain 中值是 nil。
    HandleChain(support ConsenterSupport, metadata *cb.Metadata) (Chain, error)
}
```

当排序服务节点接收到创建通道的请求时，会根据链创世区块里配置的通道类型创建新的通道，调用的是不同通道类型的 HandleChain 函数，返回能够处理链上交易的 Chain

对象。传入的参数 `support ConsenterSupport` 提供交易过滤、交易切割、区块签名等功能。参数 `metadata *cb.Metadata` 可以保存一些跟排序服务相关的元数据（`BlockMetadataIndex_ORDERER`），比如保存 Kafka 最新的偏移（`Offset`）。元数据的写入和读取是由不同的排序类型服务完成的，不同类型的排序服务有不同的处理逻辑，比如 `solo` 就不需要这个元数据信息。需要注意的是，元数据会写入到区块数据中，不同的排序服务区块元数据是不一样的，所以排序服务是不能动态切换的。

## 2. 链消息处理的接口

排序服务接收到某个通道上的交易后会提交给 Chain 处理，需要实现的接口如下所示：

```
type Chain interface {
    // 成功接收消息返回 true，否则返回 false
    Enqueue(env *cb.Envelope) bool
    // 发生错误时，会返回报错的通道，这对 Deliver 客户端来说很重要，在没有达成最新共识时可以
    // 终止客户端的等待
    Errored() <-chan struct{}
    // 用于分配 Chain 的各种资源，并保持相关的最新状态。通常情况，包括从排序服务中读取资源，
    // 将消息传递给区块进行拆分，以及将区块结果写入账本 Start()
    // 用于释放给 Chain 分配的资源
    Halt()
}
```

接口 Chain 接收交易请求并进行排序，生成最终的区块。接口 Chain 提供了可以提交消息进行排序的方法。在实现这个接口的时候，需要把排好序的交易通过 `blockcutter.Receiver` 进行交易分割，最后写到账本中。交易的分割有两种模式。

1) 交易先进入消息流中，它在消息流中是有序的，消息流中的交易分割到不同的区块里，最后写入到账本中。`solo` 和 `Kafka` 都是这种模式。

2) 交易分割到不同的区块中，区块是有序的，最后写入到账本中。`sbft` 是这种模式。

## 3. 增加新的排序服务支持

排序服务的支持需要配置文件的支持和配置文件参数的识别。在生成创世区块的配置文件 `configtx.yaml` 的参数 `Orderer.OrdererType` 中，添加新的排序服务类型，比如 `newconsenter`。

添加了配置文件以后，排序服务节点还需要能够识别新增加的参数，在 `initializeMultiChainManager` 的 `consenters` 里增加一个类似 `newconsenter` 的映射：

```
consenters["newconsenter"] = newconsenter.New()
```

在 `newconsenter` 里实现新的排序和共识算法。注意：配置参数里配置的名称要和新增的名称保持一致。

如果新的排序服务需要更多的参数支持，可以参考 `Kafka` 的参数设置方法，修改相关的代码。

### 6.3.2 基于单进程的排序服务

#### 1. 创建链的实现

单进程 (Solo) 的实现方式比较简单, 利用 Golang 的并发机制内部构建一个接收消息的通道 sendChan, 然后返回处理交易信息的链 multichain.Chain:

```
func newChain(support multichain.ConsenterSupport) *chain {
    return &chain{
        batchTimeout: support.SharedConfig().BatchTimeout(),
        support:      support,
        sendChan:     make(chan *cb.Envelope),
        exitChan:     make(chan struct{}),
    }
}
```

其中, batchTimeout 是最长的区块生成间隔时间, support 辅助提供交易切割和生成区块的功能, exitChan 是服务异常的终止信号。

#### 2. 接收交易请求的实现

创建链以后, 还需要通过 Start() 启动链的处理过程。在 Solo 的实现中, 就是启动一个协程循环的接收发送到 sendChan 通道的数据, 然后进行交易的切割和区块的写入。

排序服务接收到交易请求以后, 会根据不同的排序服务类型提交给不同的链进行处理, 入口函数就是 Enqueue。Solo 类型接收到消息以后发送给 sendChan 就结束了。

#### 3. 错误处理的实现

排序服务内部如果出现异常, 会给 exitChan 发送消息, 外部程序可以读取 Errored 返回的通道, 进行异常处理。

### 6.3.3 基于 Kafka 的排序服务

基于 Kafka 的排序服务利用 Kafka 作为交易的消息队列, 实现高吞吐量的数据分发。每个通道都对应 Kafka 的一个主题 (topic), 排序服务节点在不同阶段充当不同的角色。

1) 接收交易阶段: 排序服务节点充当的是 Kafka 的生产者 (producer), 接收到交易后通过权限检查转发给对应通道的主题。

2) 消息处理阶段: 排序服务节点充当的是 Kafka 的消费者 (consumer), 实时监听消息进行后续的处理, 生成区块或者交易分割消息等。

#### 1. 创建链的实现

在基于 Kafka 的排序服务创建链的时候, 同样返回一个能处理交易的 multichain.Chain 对象, 实际返回的是实现了 Chain 接口的 chainImpl, 实现如下所示:

```
type chainImpl struct {
```

```

consenter commonConsenter
support multichain.ConsenterSupport

channel channel
lastOffsetPersisted int64
lastCutBlockNumber uint64

producer sarama.SyncProducer
parentConsumer sarama.Consumer
channelConsumer sarama.PartitionConsumer

// 当发生分区消耗错误时关闭通道, 否则, 它是一个开放无缓冲的通道
errorChan chan struct{}

// 当收到 Halt() 请求时关闭 channel, 与 errorChan 不同, channel 在关闭时不会重新启动
// 打开, 伴随着关闭还将触发 processMessagesToBlock 退出循环
haltChan chan struct{}

// 在 Start 重试步骤完成后关闭
startChan chan struct{}
}

```

特别注意的参数是 lastOffsetPersisted 和 lastCutBlockNumber。参数 lastOffsetPersisted 记录的是排序服务节点最近读取 Kafka 集群消息的偏移量 (offset)。Kafka 的每个分区 (partition) 都是有序的消息队列, 偏移量用来表示队列中每个消息的序列号。通过这个偏移量, 排序服务节点就能确定哪些消息是已经处理过的, 哪些是还需要后续继续处理的。这个偏移量是持久化保存在区块的 Metadata 中, 类型是 BlockMetadataIndex\_ORDERER。这样, 不同的排序服务节点读取最新的区块就能确定最新读取过的消息偏移量了。那么这个偏移量是如何更新的呢? 偏移量记录的是当前区块对应的 Kafka 分区中最后一个交易的序列号, 有两种情况会产生新的区块, 在区块中记录当前区块交易在 Kafka 中的偏移量。

#### (1) 正常的交易分割

当交易数量达到设定的最大交易数 Orderer.BatchSize.MaxMessageCount、未打包交易的大小超过设定的区块大小 Orderer.BatchSize.PreferredMaxBytes 或者新提交的交易加入后未打包交易大小超过设定的区块大小时, 会进行区块分割。由于交易是顺序处理的, 满足分割条件就会分割出一个区块, 所以一次最多会分割出两个区块, 分割出的第二个区块最多只有一个交易。同时生成多个区块, 并更新每个区块偏移量的时候可以用最后一个交易在 Kafka 中的偏移量作为最后一个区块的偏移量, 前一个交易的偏移量为上一个区块的偏移量, 用算法来描述就是:

```
offset := receivedOffset - int64(len(batches)-i-1)
```

其中, receivedOffset 为从 Kafka 集群中接收到交易。



## (2) 超时的交易分割

当距离产生上一个区块的时间间隔超过设定的最大区块间隔时间 `Orderer.BatchTimeout` 时, 会检查超时打包消息记录的区块号是否正好是下一个分割区块的序号, 如果 `ttcNumber == *lastCutBlockNumber+1` 就分割未打包的交易形成新的区块。详细的区块分割逻辑参考后面的内容。

参数 `lastCutBlockNumber` 记录的是排序服务节点这个链上最近分割区块的序号。由于不同的节点是独立打包的, 所以各个节点的时间并不会完全一致。目前的实现方案是每个节点都有一个交易打包超时的定时器, 时间到了就会产生一个超时打包消息 `KafkaMessage TimeToCut` 并提交到链对应的分区上。多个排序服务节点可能产生多个相同的超时打包消息, 每个节点都以第一个超时打包消息为准, 自动忽略后面相同区块号的超时打包消息, 这样各个节点的区块打包过程就一致了。而且当各个节点打包速度不一致时, 这也能根据 `Kafka` 里的消息独立完成。这是一种把时间同步转换成消息同步的机制。

## 2. 接收交易请求的实现

基于 `Kafka` 的排序服务节点提供两种服务角色:

1) 对记账节点和应用程序, 排序服务节点是服务端, 则提供原子的广播服务, 包括 `Broadcast` 和 `Deliver` 服务。

2) 对 `Kafka` 集群, 排序服务节点是客户端, 则接收记账节点和应用程序的交易请求并转发给 `Kafka` 集群, 再从集群获取交易, 进行打包生成区块再广播给记账节点。

排序服务节点接收请求的过程, 如图 6-5 所示。

排序服务节点启动的时候会读取配置文件 `orderer.yaml` 的地址 `General.ListenAddress` 和端口 `General.ListenPort` (如果设置了环境变量 `ORDERER_GENERAL_LISTENADDRESS` 和 `ORDERER_GENERAL_LISTENPORT`, 以环境变量为准), 在指定的地址和端口上启动服务进行监听。接收到 `Broadcast` 的 `gRPC` 请求以后, 会检查类型是否是 `HeaderType_CONFIG_UPDATE` 来判断是否是配置交易, 配置交易有两种情况。

1) **配置更新请求**: 它会转换成类型为 `HeaderType_CONFIG` 的交易请求, 转换的时候会对比配置更新的内容和当前最新的配置信息, 生成包含最新更新内容的全量配置交易请求。

2) **新链创建请求**: 它会转换成类型为 `HeaderType_ORDERER_TRANSACTION` 的交易请求, 转换的时候会读取系统链上的配置信息, 生成包含组织信息等全量配置交易请求。

排序服务节点作为 `Kafka` 集群的生产者提交请求到通道对应的 `Kafka` 集群主题和分区上, 提交成功就给发送请求方返回状态为 `Status_SUCCESS` 的 `BroadcastResponse`。需要注意的是, 新链创建请求是提交到系统链通道上进行处理的, 这个时候还没有处理新链消息的链包装对象, 需要系统链在处理的时候创建出来, 交易请求也会记录到系统链的区块里, 只是系统链的区块只在排序服务节点上能看到。提交到 `Kafka` 集群中的消息处理过程如图 6-6 所示。

图 6-6 所示的消息处理已经对细节部分进行了简化, 看起来依然比较复杂。总结一下消息处理的过程, 主要包含如下几个部分。



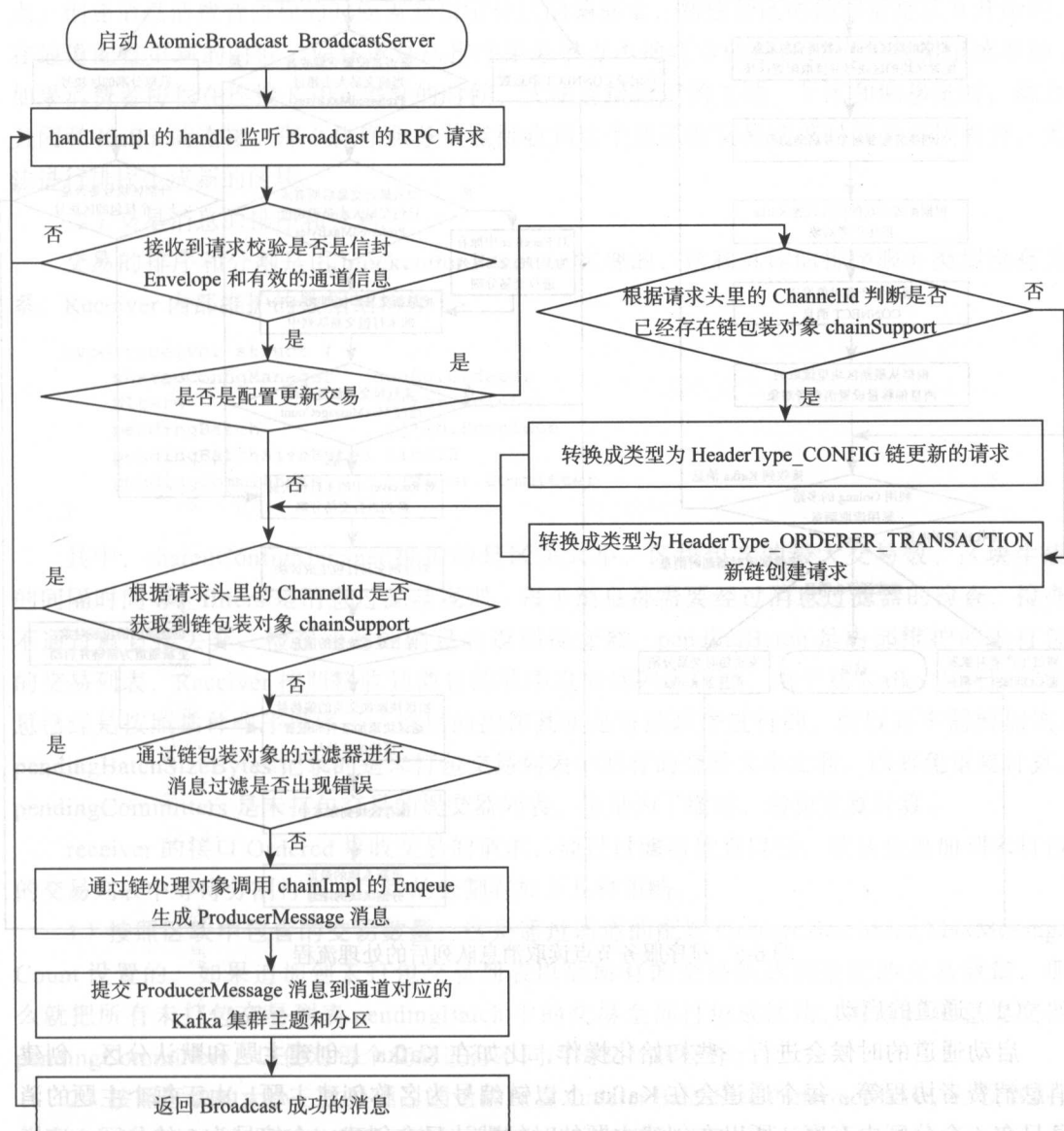


图 6-5 排序服务节点接收到请求加入队列的处理流程

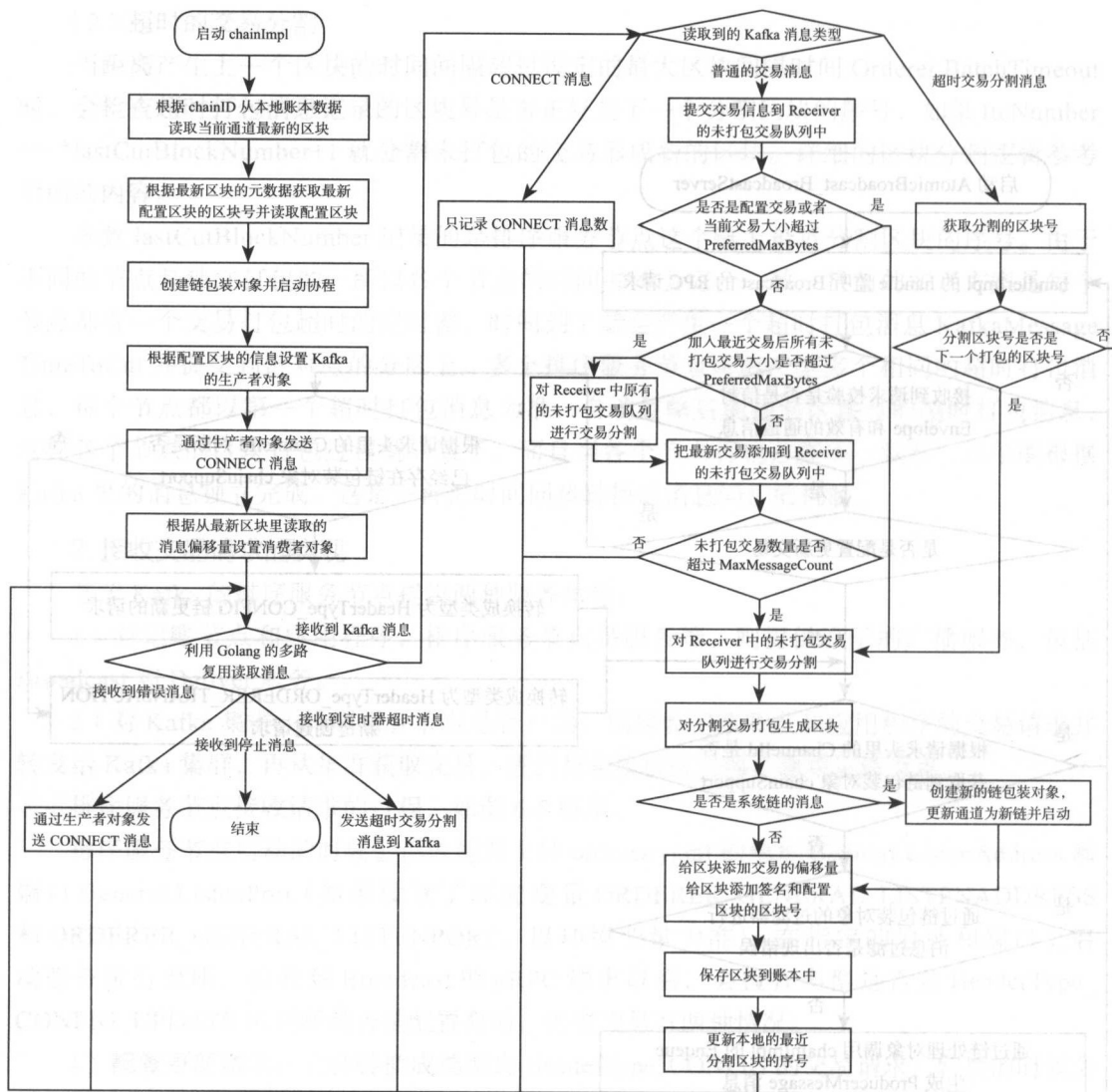


图 6-6 排序服务节点读取消息队列后的处理流程

### (1) 通道的启动

启动通道的时候会进行一些初始化操作，比如在 Kafka 上创建主题和默认分区，创建消息消费者协程等。每个通道会在 Kafka 上以链编号为名称创建主题，由于每个主题的消息只在一个分区内有序，所以在创建主题的时候默认只会创建一个编号为 0 的分区，这样从排序服务节点提交到 Kafka 的消息都是有序的。创建主题和分区的方法是给 Kafka 发送一个 CONNECT 的消息，这个消息是在系统链接收到创建链交易请求以后提交区块的过程中发送的。所有的排序服务节点都能作为消费者接收到创建链的交易请求，独立进行消息处理，也都会发送 CONNECT 消息，所以 Kafka 的同一个主题的同一个分区会有多个相同

的 CONNECT 消息, 各个排序服务节点接收到以后自动忽略即可。由于所有的排序服务节点都同时从 Kafka 上读取消息, 所以第一个接收到创建链消息的节点并不一定是提交创建链消息给 Kafka 的排序服务节点, 也不能确定第一个发送 CONNECT 消息的节点是哪个节点。创建消息消费者协程的时候需要指定分区的偏移量, 新建分区的偏移量是从 0 开始的, 在通道配置更新的时候, 偏移量是从排序服务节点本地账本的最新区块元数据里读取的。如果消费者协程在连接 Kafka 集群的时候, 无法连接指定的主题、分区和偏移量时, 就会关闭和 Kafka 的连接, 那么排序服务节点接收到这个通道的交易请求时就会直接丢弃, 无法进行排序生成新的区块。

## (2) 交易消息的排序和分割

交易的排序和分割是由 blockcutter.Receiver 实现的, 这和具体的排序服务类型没有关系。Receiver 内部维护的数据结构如下:

```
type receiver struct {
    sharedConfigManager config.Orderer
    filters              *filter.RuleSet
    pendingBatch         []*cb.Envelope
    pendingBatchSizeBytes uint32
    pendingCommitters    []filter.Committer
}
```

其中, sharedConfigManager 维护的是区块大小、区块包含的最大交易数、区块生成的间隔时间等。filters 是消息过滤器规则, 每个消息都需要经过消息过滤器的检查, 检查不通过就直接丢弃, 检查通过的消息会返回提交器。pendingBatch 是内部维护的未打包的交易列表, Receiver 按照接收到消息的顺序追加到列表后面, 由于从 Kafka 上获取的消息已经是按照某种顺序排序了, 这里的操作其实是按照顺序进行的, 所以并不需要加锁。pendingBatchSizeBytes 记录的是未打包交易列表中所有的交易大小之和, 以避免重复计算。pendingCommitters 是未打包交易的提交器列表, 也是为了缓存, 避免重复计算。

receiver 的接口 Ordered 接收交易的请求, 经过过滤器检查以后, 默认会追加到未打包的交易列表中等待分割打包。区块的分割有如下几种策略。

1) 按照区块中包含的交易数量: 这是通过通道的配置 Orderer.BatchSize.MaxMessageCount 设置的。如果追加到未打包交易列表以后所有的交易数达到设定的交易数量, 那么就把所有未打包交易列表 pendingBatch 中的交易全部打包成区块, 并返回消息提交器 pendingCommitters, 以便对每个交易进行不同的提交处理操作。

2) 按照区块的大小: 这是通过通道的配置 Orderer.BatchSize.PreferredMaxBytes 设置的。如果追加到未打包交易列表以后所有的交易大小总和达到设定的区块大小, 就需要先进行交易分割, 把原来未打包的交易列表分割以后, 再把新的交易请求追加到 pendingBatch 中。如果新交易请求的大小已经超过了区块大小的限制, 则会单独打包成一个区块。

3) 按照区块的间隔时间: 这是通过通道的配置 Orderer.BatchTimeout 设置的。在接收到新区块的第一个交易以后会启动一个定时器, 若在超时时间之内还没有足够的交易生成

新的区块，就发送一个 `KafkaMessageTimeToCut` 消息主动触发交易的分割，每个排序服务节点接收到超时交易分割消息以后，把所有未打包的交易列表 `pendingBatch` 中的交易全部打包生成新的区块。

4) 按照区块中包含的交易类型：这是内置规则，配置交易和普通交易存放在不同的区块中。配置交易记录了组织、排序服务配置等内容，是单独打包成区块并添加到账本数据中的。创世区块就是一个配置区块，如果对配置信息进行了修改，还会创建新的配置区块，每个配置区块都包含了全量的配置信息，并把配置区块的区块号保存在后续生成的普通区块中。这样，从普通区块就能快速地找到最新的配置区块，不用遍历所有的区块。配置交易的判断是通过配置消息过滤器（`configFilter`）来实现的，交易能通过 `Isolated` 函数的返回值识别出来是否配置交易。

交易分割是对排序服务节点 `Receiver` 维护的未打包交易列表 `pendingBatch` 进行操作，如果在此过程中排序服务节点出现异常，则重新启动服务也能从异常中恢复。排序服务节点启动的时候会从本地的账本数据中读取最新的区块，读取元数据里保存的 `LastOffsetPersisted`，初始化读取 `Kafka` 的消费者对象，重构 `pendingBatch` 列表进行交易分割。

### (3) 系统链和普通链的处理

系统链和普通链的处理过程基本类似，系统链是为了维护多链而存在的，记录了联盟链的组织信息等内容，创建新链的请求也是提交到系统链中处理的。系统链创建并启动新链以后的配置更新就由普通链自行处理，所以系统链只是记录普通链的初始配置信息，最新普通链的配置信息还由普通链自行维护。

### (4) 重复消息的处理

有两种类型的重复消息，一种是创建主题和分区的 `CONNECT` 消息，另外一种是超时设置的交易分割消息 `KafkaMessageTimeToCut`。这两种消息都由每个排序服务节点独立生成的，重复的消息会自动过滤。发送完 `CONNECT` 消息就完成了它的作用，所以接收到实际的 `CONNECT` 消息都不会有业务逻辑的处理。超时分割消息的过滤方法是通过内部记录的指针 `lastCutBlockNumber` 来识别的，具体的过程前面已经介绍过。

## 3. 基于 `Kafka` 排序服务的最佳实践

本节提供一些基于 `Kafka` 排序服务的参考部署。

### (1) `Kafka` 和 `Zookeeper` 节点数的选择

假设 `Kafka` 和 `Zookeeper` 的节点数分别用  $K$  和  $Z$  来表示。

1)  $K$  最少需要 4 个节点，才可以在 1 个节点宕机以后还能继续提交交易和排序，并且创建新的通道，后面的步骤会说明为什么最少是 4 个节点。

2)  $Z$  选择 3、5 或 7 个节点都可以。选择奇数个节点可以避免脑裂，1 个节点会存在单点问题，7 个以上的节点就太多了。

### (2) 创建创世区块

编辑 `configtx.yaml` 文件，主要修改项如下所示：

❑ `Orderer.OrdererType` 设置为 `Kafka`

❑ `Orderer.Kafka.Brokers` 设置至少 2 个 `Kafka` 的节点地址

❑ `Orderer.AbsoluteMaxBytes` 设置区块最大的字节数（不包括请求头）

### （3）配置 `Kafka` 集群

选择 `Kafka` 作为消息队列需要保证数据的一致性，需要对每个节点都进行如下的配置，如表 6-2 所示。

表 6-2 `Kafka` 节点配置

配置项	推荐配置	说明
<code>unclean.leader.election.enable</code>	false	参数 <code>unclean.leader.election.enable</code> 表示是否允许不在副本同步集合（In-Sync Replicas Set, ISR）中的节点选举为主节点（Leader），主节点负责消息的读写。 <code>Kafka</code> 数据的副本（Replica）通常设置为多个，即相同的数据在不同的节点上存储多份，以保证消息的可靠性。副本同步集合是在主节点之间的数据同步延迟时间在 <code>replica.lag.time.max.ms</code> 之内的节点列表，一条消息只有在副本同步集合中所有的节点都同步完成才算提交成功。如果允许不在副本同步集合中的节点选举为主节点，那么就可能存在数据丢失的风险。数据一致性是区块链必须满足的要求，需要设置这个参数为 false
<code>min.insync.replicas = M</code>	最小值为 2	参数 <code>min.insync.replicas</code> 代表最小可写入副本数的个数，大于最小值，一个消息才被确认写入成功，副本同步集合中的节点数少于这个值就不能写入。增加这个值，能够提供消息的可靠性，但是也会影响性能。 <code>M</code> 的值与后面的参数 <code>N</code> 有关系，需要满足 $1 < M < N$ ，大于 1 避免出现单点问题而丢失数据，小于 <code>N</code> 是出于性能的考虑，能选择的最小值是 2
<code>default.replication.factor = N</code>	最小值为 3	参数 <code>default.replication.factor</code> 代表创建主题时保存的副本数，也是新消息提交同步的节点数，即创建新通道时需要有至少 <code>N</code> 个节点同步完成数据才能创建成功。 <code>N</code> 的值需要满足 $1 < M < N < K$ 。 <code>N</code> 不要设置成和 <code>K</code> 相同的值。原因在于， <code>K</code> 是 <code>Kafka</code> 的节点数，若设置成和节点数相同的副本数，则任意一个节点出现异常就不能创建新的通道了。在满足 $1 < M < N$ 的情况下， <code>N</code> 能选择的最小值是 3。这样就有了前面的结论， <code>K</code> 能选择的最小值是 4。在这种配置下，允许当 1 个节点出现异常的情况下，还能继续创建新的通道，也能提交新交易进行排序生成新的区块
<code>message.max.bytes</code>	103 809 024	<code>message.max.bytes</code> 代表提交给 <code>Kafka</code> 节点消息的最大字节数，需要满足 <code>Orderer.AbsoluteMaxBytes &lt; message.max.bytes</code> ，否则会导致消息提交失败，其中， <code>Orderer.AbsoluteMaxBytes</code> 是在配置文件 <code>configtx.yaml</code> 中设置的区块最大字节数
<code>replica.fetch.max.bytes</code>	103 809 024	<code>replica.fetch.max.bytes</code> 代表从 <code>Kafka</code> 节点中获取消息的最大字节数，需要满足 <code>message.max.bytes &lt;= replica.fetch.max.bytes</code> ，否则会数据复制同步失败

### （4）连接 `Kafka` 节点出现异常时的重试设置

排序服务节点和 `Kafka` 节点连接设置了一些重试机制，主要分为如下几类。

❑ 快速重试的设置：`Kafka.Retry.ShortInterval` 和 `Kafka.Retry.ShortTotal`，用来控制快速重试的时间间隔和总时间。

❑ 升级重试的设置：`Kafka.Retry.LongInterval` 和 `Kafka.Retry.LongTotal`，用来控制升级



重试的时间间隔和总时间。

- ❑ 网络重试的设置: `Kafka.NetworkTimeouts.DialTimeout`、`Kafka.NetworkTimeouts.ReadTimeout` 和 `Kafka.NetworkTimeouts.WriteTimeout`, 用来控制网络状况的设置, 包括连接超时时间、读取数据超时时间、写入数据超时时间。
- ❑ 元数据重试的设置: `Kafka.Metadata.RetryBackoff` 和 `Kafka.Metadata.RetryMax`, 元数据记录了当前可用的 Kafka 节点地址及其分区等信息。
- ❑ 生产者重试的设置: `Kafka.Producer.RetryBackoff` 和 `Kafka.Producer.RetryMax`, 生产者写入数据时等待节点稳定的退避间隔时间和重试次数。
- ❑ 消费者重试的设置: `Kafka.Consumer.RetryBackoff`, 消费者读取数据时等待节点稳定的退避间隔时间。

快速重试和升级重试的关系和策略如下所示。

快速重试是在出现连接异常的情况下的重试策略, 间隔时间和重试的总时间较短。升级重试是在快速重试失败的情况下, 更长时间间隔的重试。排序服务节点连接 Kafka 节点失败的重试策略如图 6-7 所示。

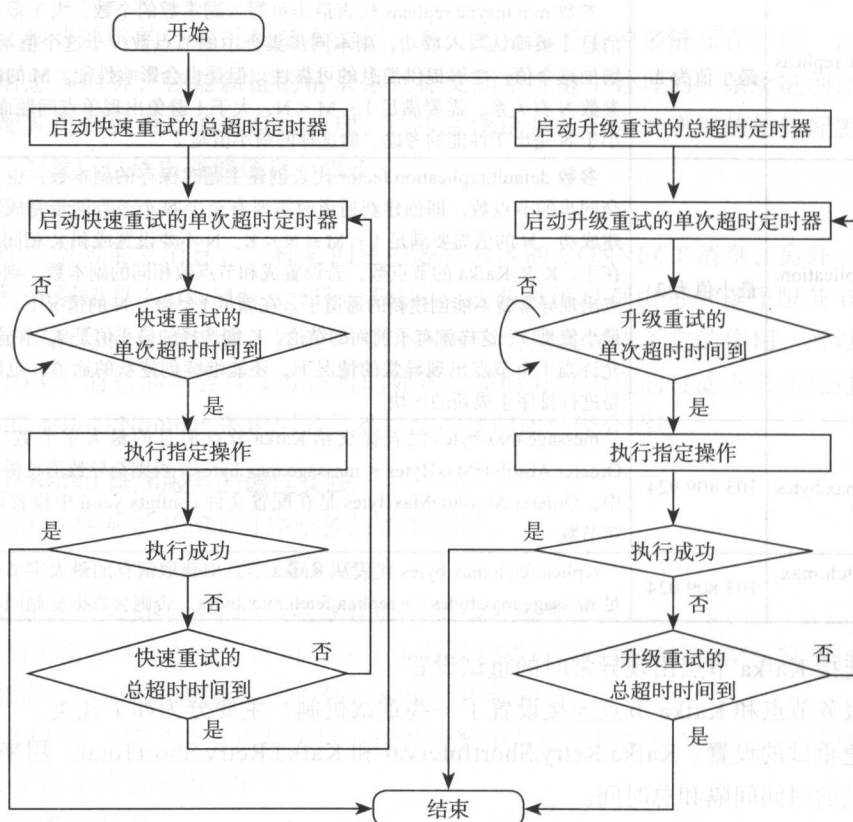


图 6-7 排序服务节点连接 Kafka 节点失败的重试策略

在配置文件 orderer.yaml 中重试设置项如表 6-3 所示。

表 6-3 orderer.yaml 中的重试设置项

配置项	推荐配置	说明
Kafka.Retry.ShortInterval	5s	连接 Kafka 节点失败后的快速重试间隔时间
Kafka.Retry.ShortTotal	10m	连接 Kafka 节点失败后的快速重试总时间
Kafka.Retry.LongInterval	5m	连接 Kafka 节点失败后的升级重试间隔时间
Kafka.Retry.LongTotal	12h	连接 Kafka 节点失败后的升级重试总时间
Kafka.NetworkTimeouts.DialTimeout	10s	连接 Kafka 节点的超时时间
Kafka.NetworkTimeouts.ReadTimeout	10s	读取 Kafka 数据的超时时间
Kafka.NetworkTimeouts.WriteTimeout	10s	写入 Kafka 数据的超时时间
Kafka.Metadata.RetryBackoff	250s	获取 Kafka 元数据时等待节点稳定的退避间隔时间
Kafka.Metadata.RetryMax	3	获取 Kafka 元数据时等待节点稳定的重试次数
Kafka.Producer.RetryBackoff	100s	写入 Kafka 数据时等待节点稳定的退避间隔时间
Kafka.Producer.RetryMax	3	写入 Kafka 数据时的重试次数
Kafka.Consumer.RetryBackoff	2s	读取 Kafka 数据时等待节点稳定的退避间隔时间

#### (5) 排序服务节点和 Kafka 节点之间的安全传输

在生产环境中，对于排序服务节点和 Kafka 节点之间的加密传输，需要同时在传输的两端进行设置。

但目前 Hyperledger Fabric 1.0 对于 Kafka TLS 的实现存在漏洞，所以若想使用 Kafka TLS 必须修改源码并重新编译生成 orderer 镜像。

##### 1) 修改源码，重新生成 orderer 镜像。

源码 fabric/orderer/kafka/config.go 第 39 行的 X509KeyPair 参数错误地将证书路径当成了证书内容来使用，45 行同样如此。

所以将这部分源码修改为：

```
if brokerConfig.Net.TLS.Enable {
    // 使用方法 LoadX509KeyPair 创建公私钥对
    keyPair, err := tls.LoadX509KeyPair(tlsConfig.Certificate, tlsConfig.PrivateKey)
    if err != nil {
        logger.Panic("Unable to decode public/private key pair====:", err)
    }
    // 创建根 CA 池
    rootCAs := x509.NewCertPool()
    for _, certificate := range tlsConfig.RootCAs {
        // 现根据路径获取证书
        caCert, err := ioutil.ReadFile(certificate)
        if err != nil {
            logger.Panic("Unable to load CA cert file.")
        }
        if !rootCAs.AppendCertsFromPEM(caCert) {
            logger.Panic("Unable to parse the root certificate authority
            certificates (Kafka.Tls.RootCAs)===")
        }
    }
}
```



```

    }
}
brokerConfig.Net.TLS.Config = &tls.Config{
    Certificates: []tls.Certificate{keyPair},
    RootCAs:      rootCAs,
    MinVersion:    tls.VersionTLS12,
    MaxVersion:    0, // 最新支持的 TLS 版本
}
}

```

## 2) 生成 Kafka TLS 证书。

假设有 3 个 Kafka 节点，分别是 kafka0、kafka1、kafka2，以及 1 个 orderer 节点。生成证书脚本如下：

```

# 利用 OpenSSL 工具生成 CA 的私钥和证书
openssl genrsa -out ca.key 2048
printf "CN\BJ\nBJ\nPS\nPS\nps.com\n\n" | openssl req -x509 -new -nodes -key
ca.key -days 3650 -out ca.crt

# 利用 OpenSSL 工具生成 Kafka 节点的私钥
openssl genrsa -out server.key 2048

# 利用 OpenSSL 工具生成 Kafka 节点的证书
printf "CN\BJ\nBJ\nPS\nPS\nkafka0\n\n\n" | openssl req -new -key server.key
-out kafka0.csr
openssl x509 -req -in kafka0.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out
kafka0.crt -days 3650

printf "CN\BJ\nBJ\nPS\nPS\nkafka1\n\n\n" | openssl req -new -key server.key
-out kafka1.csr
openssl x509 -req -in kafka1.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out
kafka1.crt -days 3650

printf "CN\BJ\nBJ\nPS\nPS\nkafka2\n\n\n" | openssl req -new -key server.key
-out kafka2.csr
openssl x509 -req -in kafka2.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out
kafka2.crt -days 3650

# 利用 OpenSSL 工具生成客户端证书
openssl genrsa -out client.key 2048
printf "CN\BJ\nBJ\nPS\nPS\nclient\n\n\n" | openssl req -new -key client.key
-out client.csr
openssl x509 -req -in client.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out
client.crt -days 3650

# 利用 OpenSSL 工具转换证书格式
openssl pkcs12 -export -in kafka0.crt -inkey server.key -out server.pk12 -name
kafka0 -passout pass:test1234
printf "test1234\ntest1234\nY\n\n" | keytool -importkeystore -deststorepass
test1234 -destkeypass test1234 -destkeystore kafka0.keystore.jks -srckeystore

```

```

server.pk12 -srcstoretype PKCS12 -srcstorepass test1234 -alias kafka0

openssl pkcs12 -export -in Kkafka1.crt -inkey server.key -out server.pk12 -name
kafka1 -passout pass:test1234
printf "test1234\ntest1234\nY\n\n" | keytool -importkeystore -deststorepass -
test1234 -destkeypass test1234 -destkeystore kafka1.keystore.jks -srckeystore
server.pk12 -srcstoretype PKCS12 -srcstorepass test1234 -alias kafka1

openssl pkcs12 -export -in kafka2.crt -inkey server.key -out server.pk12 -name
kafka2 -passout pass:test1234
printf "test1234\ntest1234\nY\n\n" | keytool -importkeystore -deststorepass
test1234 -destkeypass test1234 -destkeystore kafka2.keystore.jks -srckeystore
server.pk12 -srcstoretype PKCS12 -srcstorepass test1234 -alias kafka2

# 导入签名证书到 JKS 中
printf "test1234\ntest1234\nY\n\n" | keytool -keystore server.truststore.jks
-alias CARoot -import -file ca.crt
printf "test1234\n\n" | keytool -keystore server.truststore.jks -alias kafka0
-import -file kafka0.crt
printf "test1234\n\n" | keytool -keystore server.truststore.jks -alias kafka1
-import -file kafka1.crt
printf "test1234\n\n" | keytool -keystore server.truststore.jks -alias kafka2
-import -file kafka2.crt
printf "test1234\n\n" | keytool -keystore server.truststore.jks -alias client
-import -file client.crt

```

其中:

① printf 为后边 OpenSSL 或 keytool 命令的输入参数, 在执行命令时去掉这部分, 可按照提示手动输入相关参数。

② test1234 为密码, 可换成自己的正式密码, 此处用于测试。

3) 服务配置。生成证书后, 需配置 Kafka 和 orderer 服务。

① orderer 服务配置。

orderer 作为 Kafka 服务的客户端, 需要配置私钥、证书及 CA。具体参数配置如下所示:

```

- ORDERER_KAFKA_TLS_ENABLED=true
- ORDERER_KAFKA_TLS_PRIVATEKEY=xxx/client.key
- ORDERER_KAFKA_TLS_CERTIFICATE=xxx/client.crt
- ORDERER_KAFKA_TLS_ROOTCAS=[xxx/ca.crt]

```

其中 client.key、client.crt、ca.crt 为上一步生成的文件。

② Kafka 服务配置。对于每一个 kafka 都有如下配置:

```

- KAFKA_LISTENERS=PLAINTEXT://:8092,SSL://:9092
- KAFKA_ADVERTISED_LISTENERS=PLAINTEXT://:8092,SSL://:9092
- KAFKA_SSL_CLIENT_AUTH=required
- KAFKA_SSL_KEYSTORE_LOCATION=xxx/kafka0.keystore.jks
- KAFKA_SSL_TRUSTSTORE_LOCATION=xxx/server.truststore.jks
- KAFKA_SSL_KEY_PASSWORD=test1234

```

```

- KAFKA_SSL_KEYSTORE_PASSWORD=test1234
- KAFKA_SSL_TRUSTSTORE_PASSWORD=test1234
- KAFKA_SSL_KEYSTORE_TYPE=JKS
- KAFKA_SSL_TRUSTSTORE_TYPE=JKS
- KAFKA_SSL_ENABLED_PROTOCOLS=TLSv1.2,TLSv1.1,TLSv1
- KAFKA_SSL_INTER_BROKER_PROTOCOL=SSL

```

其中 kafka0.keystore.jks、server.truststore.jks 为上一步生成的文件，test1234 为上一步使用的密码。对于不同的 Kafka 服务要将 KAFKA\_SSL\_KEYSTORE\_LOCATION 设置为对应的文件。

#### 4) 启动服务并执行交易。

#### (6) Kafka 节点的异常处理

Kafka 提供的是可回溯的消息队列，消费者读取消息以后还可以重复读取数据，在排序服务中存放的是原始的交易请求。排序服务节点读取交易请求并生成区块以后，就不再需要这些交易信息了，所以 Kafka 里的数据是可以定期清理的。如果 Kafka 集群出现异常，并不会丢失所有的数据，只会影响还没有打包的交易请求。极端情况下，可以重构 Kafka 集群，只要初始化通道对应的主题，填充任意数据使其超过原有的偏移量即可。不用担心 Kafka 里插入数据带来的安全问题，最终记账还需要经过多重检查，比如消息类型的检查和签名验证、记账节点对背书策略的验证和交易内容的校验。

Kafka 只提供崩溃故障容错，并不提供拜占庭容错。就是说，目前版本并不能防止恶意节点攻击。

### 6.3.4 链消息过滤器

排序服务节点定义了一些规则来对消息进行过滤，每个过滤器处理后的状态分为 3 种。

□ 转发 (Forward): 不确定消息是否合法，转发给下一个过滤器进行处理；

□ 接收 (Accept): 确认消息合法，调用本过滤器的规则进行处理；

□ 拒绝 (Reject): 该消息是非法消息，不进行下一步的处理。

图 6-8 所示为系统链中消息过滤器的过滤顺序。

图 6-9 所示为通道中标准消息过滤器的过滤顺序。

各个过滤器的解释如下。

1) 空消息过滤器 (emptyRejectRule): 空消息过滤器会检查信封的有效载荷是否为空，如果为空就返回拒绝，不进行后续处理，否则转发给下一个过滤器处理。

2) 消息最大字节过滤器 (maxBytesRule): 消息最大字节过滤器会检查信封的大小 (包括有效载荷和签名) 是否超过消息的最大字节数，如果超过就返回拒绝，不进行后续处理，否则转发给下一个过滤器处理。

3) 消息签名验证过滤器 (sigFilter): 消息签名验证过滤器会检查信封的签名是否满足策略，如果不满足就返回拒绝，不进行后续处理，否则转发给下一个过滤器处理。

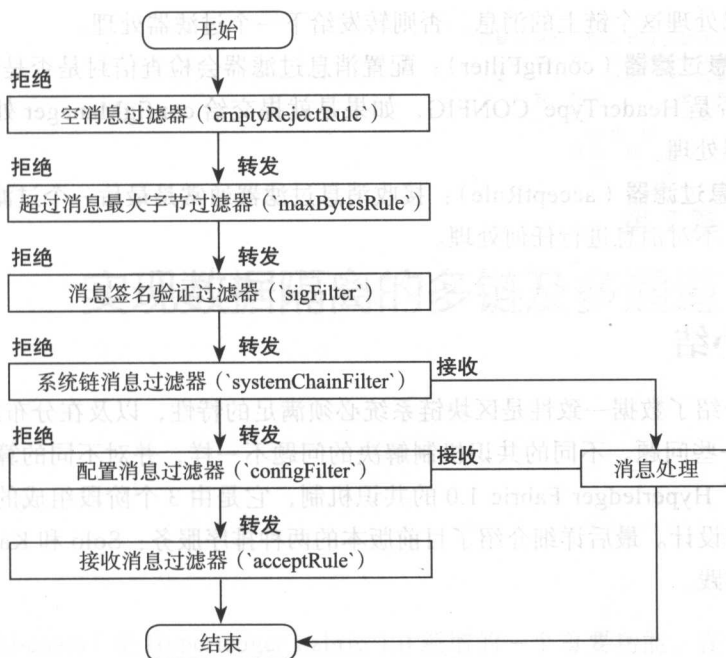


图 6-8 系统链的消息过滤流程

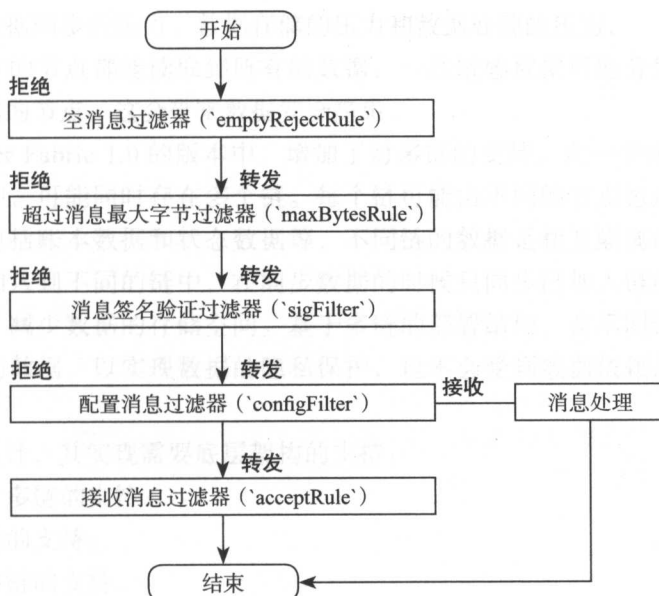


图 6-9 普通链的消息过滤流程

4) 系统链消息过滤器 (systemChainFilter): 系统链消息过滤器会检查信封是否是系统链的消息, 即检查类型是否是 HeaderType\_ORDERER\_TRANSACTION, 如果是就会创建

一个新的通道以处理这个链上的消息，否则转发给下一个过滤器处理。

5) 配置消息过滤器 (configFilter)：配置消息过滤器会检查信封是否是通道配置信息，即检查类型是否是 HeaderType\_CONFIG，如果是就提交给 configManager 处理，否则转发给下一个过滤器处理。

6) 接收消息过滤器 (acceptRule)：接收消息过滤器通常是最后一个过滤器，默认的操作是返回接收，不对消息进行任何处理。

## 6.4 本章小结

本章首先介绍了数据一致性是区块链系统必须满足的特性，以及在分布式系统环境下，可能会遇到的一些问题。不同的共识机制解决的问题不一样，并对不同的算法类型进行对比。然后介绍了 Hyperledger Fabric 1.0 的共识机制，它是由 3 个阶段组成的，每个节点都是可插拔的架构设计。最后详细介绍了目前版本的两种排序服务：Solo 和 Kafka，以及它们的实现和最佳实践。

## 实现数据隔离的多链及多通道

多链（multi-chain）是 Hyperledger Fabric 1.0 新增的一个重要功能。在 0.6 版中，所有的节点都属于一个链，所有的节点都会同步相同的数据，这会带来几个问题：

- 随着业务量的增加，数据会越来越大，每个节点都会同步和存储一些不必要的数据，这增加了数据同步的压力、数据存储的压力和数据处理的压力；
- 网络中所有的节点都能读取到所有的数据，一些敏感数据可能分发给其他不应该访问这些数据的节点，这会带来数据安全隐患。

在 Hyperledger Fabric 1.0 的版本中，增加了对多链的支持。在一个由很多 Peer 节点组成的区块链网络中，可能同时存在多个链。每个链可能由不同的节点组成，这些节点维护着相同的数据，包括账本数据和状态数据等，不同链的数据是相互隔离的。一个节点根据应用需求，可以加入到不同的链中，在同步数据的时候只同步已加入链的数据，这能减少数据同步的时间，减少数据的存储空间。基于多链的部署结构，在不同链上运行的智能合约访问的是不同的数据，以实现数据的隐私保护，也不会受到数据依赖的限制，提高了并行处理的效率。

多链是全局设计，其实现需要底层架构的支持：

- 数据存储对多链的支持；
- 链码对多链的支持；
- 多通道对多链的支持。

下面我们分开来看每个部分是如何支持多链实现的。

## 7.1 数据存储对多链的支持

数据存储包含账本数据、索引数据、状态数据和历史数据等几个部分，记账节点包含所有的数据，排序节点只包含账本数据及其索引数据，不包含状态数据及其历史数据。

### 7.1.1 账本数据

记账节点和排序节点都会存储账本数据，即区块文件。

前面的章节已经介绍过，记账节点的账本数据是基于文件系统存储的，每个链的账本数据存储在不同的目录下。只有属于某个链，才会存在以这个链的通道命名的账本目录。

记账节点的账本数据存储目录一般是 `/var/hyperledger/production/ledgersData/chains/chains/`，其中 `/var/hyperledger/production` 可以通过 `core.yaml` 文件中的 `peer.fileSystemPath` 选项指定，后面的 `ledgersData/chains/chains` 是记账节点中固定的目录后缀。下面是一个记账节点的账本数据目录结构：

```
root@peer0:/var/hyperledger/production/ledgersData/chains/chains# tree .
```

```
.
|-- businesschannel
|   |-- blockfile_000000
|   |-- pocchannel
|       |-- blockfile_000000
```

```
2 directories, 2 files
```

这个目录下面有两个目录：`businesschannel` 和 `pocchannel`。它们代表的是两个通道，也就是两个链的数据，每个链现在只有一个区块文件，`blockfile_` 是文件名中固定的前缀，`000000` 是固定的 6 位占位符，下一个文件名会依次递增。从这个目录结构可以看到，记账节点在底层账本数据存储的时候就对不同链的数据进行了隔离。

排序节点会存储所有链的账本数据，排序节点除了可以选择序列化区块文件的格式外，还支持 JSON 文件格式和内存数据结构的账本数据，后面两种都只在测试环境下使用。序列化区块文件和 JSON 文件格式区块文件的存储目录一般是 `/var/hyperledger/production/orderer/chains`，其中，`orderer/chains` 是固定的目录后缀。同样地，不同链的账本数据存储在以通道名称为目录名称的目录中，以实现不同链账本数据的物理隔离。内存数据的账本数据没有持久化的存储，不同链的账本数据存储在不同的数据结构中。

### 7.1.2 索引数据

记账节点和排序节点都会给账本数据建立索引，不同的是排序节点只会建立以 `BlockNum` 为属性的索引。

索引文件存储的目录是 `/var/hyperledger/production/ledgersData/chains/index`，其中，`ledgersData/chains/index` 是记账节点上固定的目录后缀，排序节点上的目录后缀是 `orderer/index`。



下面是一个记账节点上的索引数据的目录：

```
root@peer0:/var/hyperledger/production/ledgersData/chains/index# tree
.
|-- 000002.ldb
|-- 000007.log
|-- CURRENT
|-- LOCK
|-- LOG
`-- MANIFEST-000008
```

```
0 directories, 6 files
```

索引数据是存储在 LevelDB 数据库里的，数据库的类型目前是可选的。LevelDB 是持久化的 K-V 数据库，在保存索引的时候会加上 ledgerid 作为前缀，当然生成的组合键在构造的时候是要先转换成 []byte 数组的。由于索引数据存储在同一个数据库中，所以对于不同链的数据，索引数据的实现是逻辑隔离的，并非是物理隔离的。

### 7.1.3 状态数据

排序节点不需要查询具体的交易信息和状态数据，也不会存储状态数据及其历史数据。

Peer 节点上状态数据存储的目录是 /var/hyperledger/production/ledgersData/stateLeveldb，其中，ledgersData/stateLeveldb 是固定的后缀：

```
root@peer0:/var/hyperledger/production/ledgersData/stateLeveldb# tree
.
|-- 000002.ldb
|-- 000007.log
|-- CURRENT
|-- LOCK
|-- LOG
`-- MANIFEST-000008
```

```
0 directories, 6 file
```

状态数据也是基于 K-V 存储的，同一个节点的状态存储在同一个数据库中，没有进行物理隔离。和索引数据不同的是，状态数据是和 chaincodeID 相关的，不同 chaincodeID 的数据是逻辑隔离的，而 chaincodeID 同样是以 chainID 为前缀进行了逻辑隔离。

### 7.1.4 历史数据

历史数据存储的目录是 /var/hyperledger/production/ledgersData/historyLeveldb，其中，ledgersData/historyLeveldb 是固定的后缀：

```
root@peer0:/var/hyperledger/production/ledgersData/historyLeveldb# tree
.
|-- 000002.ldb
```

```

|-- 000010.ldb
|-- 000012.log
|-- CURRENT
|-- LOCK
|-- LOG
|-- LOG.old
|-- MANIFEST-000013
`-- level-party.sock

```

```
0 directories, 9 files
```

历史数据目前内置的数据库是 LevelDB，也是不可替换的。记录的是状态数据的历史记录，同状态数据一样，通过在构建 chaincodeID 的时候增加 ChainID 前缀来逻辑隔离不同链的数据。

## 7.2 链码对多链的支持

链码是 Hyperledger Fabric 1.0 提供的智能合约方案，实现了交易的模拟执行。链码从多个纬度对多链提供了支持，比如链码的生命周期管理、链码和背书节点的通信、链码的部署方法等。

### 7.2.1 链码的生命周期管理

智能合约在 Hyperledger Fabric 1.0 上称为链码 (Chaincode)，是独立运行的应用程序，只接收启动它的背书节点的指令，执行指定的业务逻辑。在多链的情况下，同一个智能合约可能会在不同的链上运行。为了重用智能合约代码，智能合约的部署拆分成了安装和实例化两个步骤，安装只是把链码的源代码序列化后和链码名称、版本等封装成 ChaincodeDeploymentSpec 保存到 Peer 节点上，链码安装跟具体的链没有关系，也不需要 ChainID 参数。

换个说法，不同链的链码是没有隔离的，也就是说，在一个链安装的链码可能和另外一个链的链码产生冲突。链码安装的时候会检查是否存在相同名称和版本的链码，如果不同的上层应用同时都部署了相同的链码和版本，可能存在一个链的链码安装成功，另外一个链的链码安装失败的情况。

实例化和链码升级，是在指定的链上操作的，实际过程分为两个步骤，第一步是调用系统链码 LSCC 的部署操作，通过 LSCC 把链码计算哈希后生成的 ChaincodeData 存放在状态数据库中；第二步是从文件系统中读取保存的链码源码，生成镜像后执行初始化操作。链码操作包括初始化和调用，它们都是在具体链上操作的，链码镜像的命名规则是：

```
NetworkID-PeerID-ChaincodeName-ChaincodeVersion-SHA256(ChainID)
```

链码镜像名称的最后一部分是对 ChainID 计算 SHA256 哈希后再转换成十六进制的字

符串，在逻辑上不同链的链码会有不同的镜像名称。启动的链码容器命名和镜像一样，只是会把“:”替换成“\_”。这样，不同链的链码执行是可以在不同的环境中隔离的。不过，实际在启动链码容器的时候并没有指定 ChainID 这个参数，就是说目前不同链上相同链码是运行在同一个容器中的。但即使运行在相同的链码容器中，也会通过 ChainID 进行逻辑隔离。详细的通信机制见下一节的介绍。

### 7.2.2 链码和背书节点的通信

链码容器启动以后，会和启动它的背书节点建立 gRPC 连接。应用程序或者命令行通过 gRPC 连接给背书节点发送请求，背书节点校验通过后会通过链码和背书节点建立的 gRPC 连接将请求发送给链码去执行。链码的执行本身是和具体链无关的，链码容器也不会本地保存任何数据，是一个无状态的执行环境。需要访问或者写入状态数据时，则通过建立好的 gRPC 连接发送请求给背书节点，再进行后续的业务逻辑处理。就是说，在链码这一端，是不区分链的，所以不同链才可以共用相同的链码容器。

链码容器启动的时候，和背书节点建立的 gRPC 连接没有和链相关的信息，链码通过建立好的 gRPC 连接发送给背书节点的第一个信息是：ChaincodeMessage\_REGISTER，内容是序列化后的 ChaincodeID，在不同链上其也可能是相同的（虽然 ChaincodeID 的命名规则是：ChaincodeName:ChaincodeVersion/ChainID，目前的命名也是没有 ChainID 标识的）。不管是多个容器执行链码，还是在同一个容器中执行链码，链码运行的结果都会通过 gRPC 发送给背书节点，背书节点怎么知道是哪个链上的操作呢？

链码运行是以交易号作为标识的，客户端发起 Proposal 请求时需要指定发送到哪个链上，背书节点会把交易号和链标识进行关联，再把消息发送给链码去执行，接收到链码返回的请求就知道属于哪个链了。背书节点的内部维护了一个运行中的链码映射表，键是 ChaincodeID，值是封装了 Handler 的链码运行时环境 chaincodeRTEnv，就是同一个链码都对对应同一个链码运行时环境，负责处理链码发送过来的消息。Handler 内部维护了一个交易的上下文映射表 txCtxs，键是交易号 txid，值是 transactionContext 的结构体，如下所示：

```
type transactionContext struct {
    chainID      string
    signedProp   *pb.SignedProposal
    proposal     *pb.Proposal
    responseNotifier chan *pb.ChaincodeMessage

    // 记录范围查询的迭代器
    queryIteratorMap map[string]commonledger.ResultsIterator

    txsimulator      ledger.TxSimulator
    historyQueryExecutor ledger.HistoryQueryExecutor
}
```

其中，chainID 就是链的标识，通过 txsimulator 对不同链的状态数据库进行操作，实现

不同链数据处理的逻辑隔离。交易上下文映射表 txCtxs 在每次调用链码时都会更新，链码容器启动后就会给 Handler 发送 sendReady 消息，给每个交易创建一个映射表项。接收到链码发送过来的消息后，通过 txCtxs 能对不同的链进行操作。

### 7.2.3 链码的部署和调用

多链的实现也会对链码的部署和调用方式有影响，首先需要创建一个链，fabric-sdk-go 的接口是：

```
type FabricClient interface {
    NewChannel(name string) (Channel, error)
    CreateChannel(request CreateChannelRequest) (txn.TransactionID, error)
}

type CreateChannelRequest struct {
    // 必填 - channel 名称
    Name string
    // 必填 - 发送更新请求的 Orderer
    Orderer Orderer
    // 可选 - envelope object 包含了初始化 channel 所需的设置以及签名
    // 可通过命令行工具 configtx 来创建
    Envelope []byte
    // 可选 - 通过 package 中的 buildChannelConfig() 方法来构建 ConfigUpdate 对象
    Config []byte
    // 可选 - 使用 `config` 参数时，指定创建策略所需的签名集合
    // 详细参考 signChannelConfig() 方法
    Signatures []*common.ConfigSignature

    // InvokeChannelRequest 允许传入 TransactionID 参数
    // 该请求结构虽然包含一致性字段，但也可能删除
    TxnID txn.TransactionID
}
```

其中，NewChannel 只是在本地创建一个 Channel 对象，用来进行后续的初始化和其他操作。实际的创建链请求需要调用 CreateChannel，请求参数 CreateChannelRequest 里的 Name 是和 NewChannel 中的 name 对应的。

然后链码的部署分成两个步骤，链码安装 InstallChaincode 和链码的实例化 SendInstantiateProposal，其中 InstallChaincode 是定义在 FabricClient 中的，是和链无关的操作，SendInstantiateProposal 是定义在 Channel 中的。

```
type FabricClient interface {
    InstallChaincode(chaincodeName string, chaincodePath string, chaincodeVersion
        string, chaincodePackage []byte, targets []Peer) ([]*txn.TransactionProposal
        Response, string, error)
}

type Channel interface {
    SendInstantiateProposal(chaincodeName string, args []string, chaincodePath
        string, chaincodeVersion string, chaincodePolicy *common.SignaturePolicy
```

```
Envelope, targets []txn.ProposalProcessor) ([]*txn.
TransactionProposalResponse, txn.
TransactionID, error)
}
```

最后，在链码调用的时候需要使用创建好的 Channel 对象：

```
func InvokeChaincode(client fab.FabricClient, channel fab.Channel, targets []
apitxn.ProposalProcessor,
eventHub fab.EventHub, chaincodeID string, fcn string, args []string,
transientData map[string][]byte) (apitxn.TransactionID, error)
```

### 7.3 多通道对多链的支持

排序节点同时会给多个链提供服务，会接收到多个链提交过来的交易并形成不同链的区块。Hyperledger Fabric 1.0 采用多通道的方法来隔离不同链的数据。

发送给排序服务的节点怎么区分是哪个链的数据呢？客户端在接收到背书节点返回的执行结果后，会生成最终的交易。其实，交易里面会包含发送给背书节点的 Proposal 请求，每个 Proposal 都会包含请求头 common.Header，其定义如下：

```
type Header struct {
ChannelHeader []byte `protobuf:"bytes,1,opt,name=channel_header,json=channelHeader,proto3" json:"channel_header,omitempty"`
SignatureHeader []byte `protobuf:"bytes,2,opt,name=signature_header,json=signatureHeader,proto3" json:"signature_header,omitempty"`
}
```

其中的 ChannelHeader 是包含了通道编号的序列化字节数组，如下所示：

```
// Header 是一种通用的重播预防，包含重放签名的身份信息
type ChannelHeader struct {
Type int32 `protobuf:"varint,1,opt,name=type" json:"type,omitempty"`
// 协议版本信息
Version int32 `protobuf:"varint,2,opt,name=version" json:"version,omitempty"`
// 发件人创建消息的本地时间
Timestamp *google_protobuf.Timestamp `protobuf:"bytes,3,opt,name=timestamp"
json:"timestamp,omitempty"`
// ChannelId 用于表示消息绑定 channel 的标识
ChannelId string `protobuf:"bytes,4,opt,name=channel_id,json=channelId"
json:"channel_id,omitempty"`
// 端到端的唯一标识符
// - 高级设置，如用户终端或 SDK
// - 传递给背书节点（用于唯一性检查）
// - header 会随消息一直传递，将被 committer 获取（也用于唯一性检查）
// - 会存入账本
TxId string `protobuf:"bytes,5,opt,name=tx_id,json=txId" json:"tx_id,
omitempty"`
// header 中的 Epoch 的定义取决于块高度
// response 中的 Epoch 表示逻辑窗口时间。以下两种情况时，peer 节点才接受提案响应
```

```
// 1. 消息中的 epoch 与当前 epoch 匹配
// 2. 消息只在当前 epoch 中出现一次（即没有被重播）
Epoch uint64 `protobuf:"varint,6,opt,name=epoch" json:"epoch,omitempty"`
// 可以根据 header 类型进行扩展
Extension []byte `protobuf:"bytes,7,opt,name=extension,proto3"
json:"extension,omitempty"`
}
```

排序服务在接收到交易请求后，会先反序列化得到 ChannelId，在不同的通道上进行排序打包生成区块。多通道（Multi-channel）部分的内容详见第 6 章。

## 7.4 命令行和 SDK 对多链的支持

我们在前面的章节已经介绍过命令行的使用，其中可以指定一个 -C 参数代表在指定的通道上操作，在入口处提供了对多链的支持。

我们在第 10 章还会看到在超级账本提供给应用程序的 SDK 中，也提供了多链的接口。

## 7.5 关于系统链

系统链是一个特殊的链，含有系统层面全局配置区块链网络的联盟及组织信息、MSP 信息和策略信息等，只存在于排序服务中。涉及一些信息的修改，比如增加一个组织，增加排序服务节点，这些都会在系统链上增加一个配置区块。整个系统有且只有一个系统链，系统链是通过创世区块配置的，排序服务启动的时候通过 ORDERER\_GENERAL\_GENESISFILE 环境变量指定创世区块文件并创建系统链。系统链的名称可以在创建创世区块的时候通过工具 configtxgen 的 channelID 参数指定，默认的系统链名称是 testchainid，是否是系统链的判断方法就是配置区块信息里是否有联盟信息配置项。关于创世区块的详细信息也请参考前面已经介绍过的第 6 章的相关内容。

## 7.6 本章小结

在本章中，我们介绍了在 Hyperledger Fabric 1.0 中支持的多链及其内部实现。在整个区块链网络中，支持多个链同时运行是一个系统工程，涉及所有的参与方，包括应用程序、Peer 节点、排序服务节点等，如何从这些节点中动态地组建一个链，需要很多方面的支持。从业务流程上看，也涉及多个环节，包括创世区块的创建、链码的部署、链码的调用、链码的运行、交易排序、交易验证和记账等。不同的节点加入到不同的链中，还有很多权限和策略的控制。这是一种全新的架构，增加了很多的复杂度。Hyperledger Fabric 1.0 相对于 0.6 版本而言，很多比较难理解和操作的部分都跟多链和多通道相关。理解了多链的目的后，其实自然就知道为什么要这么设计了。



## 基于数字证书的成员管理服务

Hyperledger Fabric 1.0 基于 PKI 体系，生成数字证书以标识用户的身份。每个身份和成员管理服务提供商（Membership Service Provider，MSP）的编号进行关联，本章将会介绍如何对用户身份进行认证。

### 8.1 实现成员管理的 MSP

MSP (Membership Service Provider): 即成员管理服务提供商，是 Hyperledger Fabric 1.0 中引入的一个组件，其目的是抽象化各成员之间的控制结构关系。MSP 将证书颁发、用户认证、后台的加密机制和协议都进行了抽象。每个 MSP 可以定义自己的规则，这些规则包括身份的认证，签名的生成和认证。每个 Hyperledger Fabric 1.0 区块链网络可以引入一个或者多个 MSP 来进行网络管理。这样就将成员本身和成员之间的操作、规则和流程都模块化了。

#### 8.1.1 MSP 成员的验证

我们先来看一下 MSP 的成员身份及身份标识符定义：

```
type identity struct {  
    // 身份标识符  
    id *IdentityIdentifier  
    // X.509 证书  
    cert *x509.Certificate  
    // 公钥  
    pk bccsp.Key
```



```

// 所属的 MSP
msp *bccspmsp
}

type IdentityIdentifier struct {
    // MSP 标识
    Mspid string
    // 身份编号
    Id string
}

```

从上面的定义中我们可以看到，成员身份是基于标准的 X.509 证书的。利用 PKI 体系给每个成员颁发数字证书，结合所属的 MSP 进行身份认证和权限控制。**根 CA 证书** (Root Certificate) 是自签名的证书，用根 CA 证书的私钥签名生成的证书还可以签发新的证书，形成一个树型结构。**中间 CA 证书** (Intermediate Certificate) 是由其他 CA 证书签发的，也可以利用自己的私钥签发新的证书。签发证书是一个信任背书的过程，从根 CA 证书到最终用户证书形成一个**证书信任链** (Chain of Trust)。在 PKI 体系中，可以利用 CRL (Certificate Revocation List) 或者 OCSP (Online Certificate Status Protocol) 管理证书的有效性。在超级账本中，MSP 利用 PKI 的部分特性来管理证书的有效性。

1) **MSP 标识的检查**：身份证书都是和 MSP 绑定的，必须有相同的 MSP 标识才能验证证书的有效性。Peer 节点的 Gossip 通信和部分系统链码的调用都要求调用者身份和本地 MSP 标识相同，背书请求在通道管理策略验证的时候也会检查 MSP 成员是否有写入权限。

2) **证书路径的检查**：除了 MSP 标识的检查，还会对证书签名有效性进行检查，主要是证书路径的检查，校验根 CA 证书、中间 CA 证书是否有效，是否有从身份证书到可信根 CA 证书的有效路径。特别说明一下，在证书验证的时候并不校验证书的有效期，会强制设置当前时间为证书起始时间的下一秒，这确保有效期验证通过。MSP 目录下的 cacerts 和 intermediatecerts 子目录签发的证书都是有效的，证书校验的时候需要检查是否有到可信根证书的有效路径。

3) **CRL 的检查**：最后是检查证书是否被吊销，目前只支持 CRL 的方式，并不支持 OCSP。CRL 会包含在本地 MSP 的 crls 子目录下，是由 CA 证书签发的包含被吊销证书序列号的证书文件。在通道的 MSP 配置中，也会包含 CRL 列表。在更新通道配置的时候可以发布吊销的证书。

## 8.1.2 MSP 的目录结构

下面先介绍生成 MSP 目录的一些必要准备工作。

### 1. MSP 的配置说明

在每一个 Peer 节点和排序服务节点上设置 MSP 目录后，Peer 节点和排序服务节点就有了签名证书，在通道节点之间传输数据时，要验证节点的签名。

为了能够标识 MSP，每个 MSP 需要指定一个名称，如 org1、org2 等。在通道的 MSP 成员规则中可以用 MSP 名称来代表一个联盟（Consortium）、组织（Organization）或者部门（Organization Division）。若在创世区块中检测到两个 MSP 用同一个 MSP 名称，则排序服务节点将启动失败。

MSP 的默认实现是基于 X.509 证书格式的，根据 RFC5280 文档的内容，给一些 MSP 的配置参考，如表 8-1 所示。

表 8-1 MSP 配置参考

配置项	配置说明	是否必须配置
根 CA (Root of Trust) 证书	自签名的证书列表	必须配置
中间 CA (Intermediate CA) 证书	由根 CA 证书颁发的证书	可选配置
MSP 的管理员证书	有根 CA 证书或者中间 CA 证书的可验证证书路径，已配置的管理员证书有权修改通道配置	必须配置
在 MSP 有效成员的证书中包含一个部门列表	当多个组织采用相同的根 CA 证书和中间 CA 证书时，可以预留一个部门字段作为扩展，通过部门的方法验证成员的权限，参考第 3 章的相关部分	可选配置
设置证书吊销列表 (CRL, Certificate Revocation List)	列表中的证书序列号可以是根 CA 的，也可以是中间 CA 证书的	可选配置
TLS 的根 CA 证书	自签名的证书列表	必须配置
TLS 的中间 CA 证书	TLS 的中间 CA 证书由 TLS 的根 CA 证书颁发的	可选配置

节点需要进行如下的配置才能使用 MSP 进行签名或者验签：

- 1) 用于节点签名的签名密钥（目前只支持 ECDSA 密钥）。
- 2) 通过 MSP 验证是有效的 X.509 证书将作为节点证书。

怎么才能成为 MSP 的有效身份（Identity）呢？这需要同时满足如下几个条件：

- 1) 身份证书需要符合 X.509 证书标准，且有到根 CA 证书或者中间 CA 证书可验证的证书路径。
- 2) 身份证书不在证书吊销列表中。
- 3) 在 X.509 证书的 OU 字段至少包含一个在 MSP 中配置的部门。

和普通的 X.509 证书不同的是，MSP 的身份证书是没有有效期的，除非被添加到证书吊销列表中。

## 2. 生成 MSP 证书

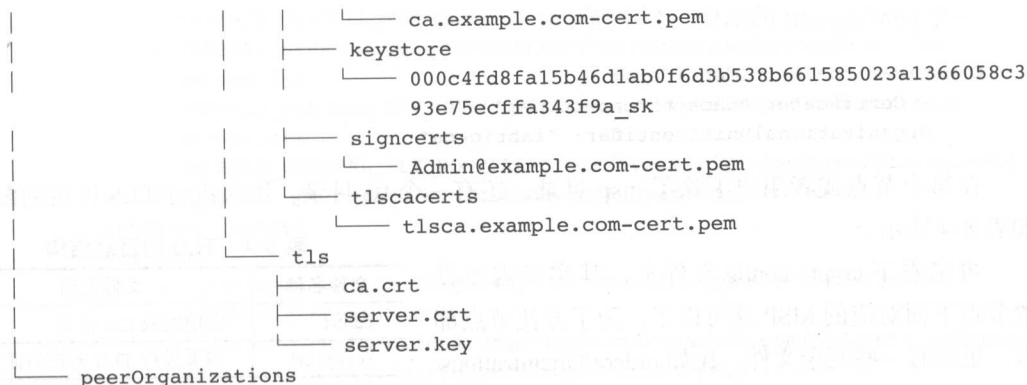
有很多的工具可以生成 X.509 证书，比如广泛使用的 OpenSSL。需要注意的是，在 Hyperledger Fabric 1.0 中，不支持包含 RSA 密钥的证书。本例中采用 cryptogen 工具来生成 MSP 证书，生成过程如下：

```
# 下载 fabric 源码，存放在 FABRIC_SRC_DIR 目录下
git clone git@github.com:hyperledger/fabric.git,
# 生成工具在 $FABRIC_SRC_DIR/hyperledger/fabric/examples/e2e_cli 目录下
cd $FABRIC_SRC_DIR/hyperledger/fabric/examples/e2e_cli
```

```
# 调用 e2e 的工具生成证书
./generateArtifacts.sh
```

执行完以上三步操作就会在目录 \$FABRIC\_SRC\_DIR/hyperledger/fabric/examples/e2e\_cli 下产生 crypto-config 文件夹。在这个 MSP 中配置了两个组织 (即 org1 与 org2), 每个组织下有两个 Peer 节点 (即 peer0 和 peer1): 一个排序服务节点, 一个 CA 节点。部分目录的结构如下所示:

```
├── ordererOrganizations
│   ├── example.com
│   │   ├── ca
│   │   │   ├── 4514ec148b0b79b58957131cf0f5d516be4b5d79b10f90f5f63ec208d71c85e1_sk
│   │   │   └── ca.example.com-cert.pem
│   │   ├── msp
│   │   │   ├── admincerts
│   │   │   │   └── Admin@example.com-cert.pem
│   │   │   ├── cacerts
│   │   │   │   └── ca.example.com-cert.pem
│   │   │   └── tlscacerts
│   │   │       └── tlsca.example.com-cert.pem
│   │   ├── orderers
│   │   │   ├── orderer.example.com
│   │   │   │   ├── msp
│   │   │   │   │   ├── admincerts
│   │   │   │   │   │   └── Admin@example.com-cert.pem
│   │   │   │   │   ├── cacerts
│   │   │   │   │   │   └── ca.example.com-cert.pem
│   │   │   │   │   ├── keystore
│   │   │   │   │   │   └── 550422b59f3ca8cf0df4a9782f1041d906b9100dab8bdf74409c445efdb96437_sk
│   │   │   │   │   ├── signcerts
│   │   │   │   │   │   └── orderer.example.com-cert.pem
│   │   │   │   │   └── tlscacerts
│   │   │   │   │       └── tlsca.example.com-cert.pem
│   │   │   │   └── tls
│   │   │   │       ├── ca.crt
│   │   │   │       ├── server.crt
│   │   │   │       └── server.key
│   │   └── tlsca
│   │       ├── 88fd39ff8fe5e0d103d804a4a3c8e172b7f6ff5adbcf6182b3cd6c1aa2d68fa5_sk
│   │       └── tlsca.example.com-cert.pem
│   └── users
│       ├── Admin@example.com
│       │   ├── msp
│       │   │   ├── admincerts
│       │   │   │   └── Admin@example.com-cert.pem
│       │   └── cacerts
```



crypto-config 目录下会生成两个目录，ordererOrganizations 和 peerOrganizations，它们分别代表排序服务节点和 Peer 节点的 MSP 配置信息。peerOrganizations 目录下有两个组织 org1.example.com 和 org2.example.com，子目录结构同 ordererOrganizations 的 example.com。我们以 ordererOrganizations 的 example.com 为例说明每个组织的目录结构，如表 8-2 所示。

表 8-2 组织的目录结构

目录名称	目录说明
ca	根 CA 证书和密钥，后缀为 .pem 的是证书，_sk 后缀的是密钥
msp	MSP 的配置，参考后面的表格
orderers 或者 peers	ordererOrganizations 目录下是 orderers，peerOrganizations 目录下是 peers，分别代表的是排序服务节点和 Peer 节点的配置
tlsca	TLS 的中间 CA 证书和密钥，后缀为 .pem 的是证书，_sk 后缀的是密钥
users	默认生成的用户配置，一般会包含 1 个管理员和 1 个普通成员

MSP 目录的说明，如表 8-3 所示。

表 8-3 MSP 的目录结构

目录文件名称	目录文件说明	是否可选
admincerts	MSP 的管理员证书	必选
cacerts	MSP 的根 CA 证书	必选
intermediatecerts	中间 CA 证书	可选
config.yaml	映射部门和证书 OU 字段的 YAML 配置文件，用 <Certificate, OrganizationalUnitIdentifier> 表示一个映射表，其中 Certificate 是包含部门信息的证书文件路径，它是相对于 MSP 根目录的路径的。OrganizationalUnitIdentifier 是 X.509 证书中的 OU 字段。详见后面的示例	可选
crls	证书吊销列表	可选
keystore	签名密钥	必选
signcerts	节点的签名证书	必选
tlscacerts	TLS 的根 CA 证书	可选
tlsintermediatecerts	TLS 的中间 CA 证书	可选

一个 config.yaml 的示例，如下所示：

```
OrganizationalUnitIdentifiers:
  - Certificate: "cacerts/cacert.pem"
  OrganizationalUnitIdentifier: "fabric-ca"
```

在每个节点或者用户下除了 msp 目录，还有一个 tls 目录，用来进行 TLS 连接的配置，如表 8-4 所示。

表 8-4 TLS 的目录结构

文件名称	文件说明
ca.crt	可信的根 CA 证书
server.crt	用来进行 TLS 连接的证书
server.key	用来进行 TLS 连接的密钥

再来看下 crypto-config 文件夹，其实只需要设置节点下面对应的 MSP 就可以了，为了方便节点部署，里面有一些冗余文件。比如 ordererOrganizations/example.com/msp 目录下子目录文件都会在 ordererOrganizations/example.com/orderers/orderer.example.com/msp 目录下，这样会增加节点 orderer.example.com 对应的签名证书和签名私钥。ordererOrganizations/example.com/orderers/orderer.example.com 其实就是管理员 ordererOrganizations/users/Admin@example.com。还有根 CA 证书和 TLS 根 CA 证书等也是冗余存储的。

### 3. 配置节点的 MSP 证书

生成 MSP 证书后，就可以启动 Peer 节点和排序服务节点了，配置说明如表 8-5 所示。

表 8-5 MSP 配置说明

配置项	Peer 节点	排序服务节点
默认节点的 YAML 配置文件名称	core.yaml	orderer.yaml
YAML 文件中的本地 MSP 路径	peer.mspConfigPath	General.LocalMSPDir
YAML 文件中的本地 MSP 名称	peer.localMspId	General.LocalMSPID
YAML 文件中是否启用 TLS	peer.tls.enabled	General.TLS.Enabled
YAML 文件中的 TLS 根 CA 证书	peer.tls.rootcert.file	General.TLS.RootCAs
YAML 文件中的 TLS 证书	peer.tls.cert.file	General.TLS.Certificate
YAML 文件中的 TLS 私钥	peer.tls.key.file	General.TLS.PrivateKey

一个节选的 docker-compose.yaml 示例文件如下所示：

```
orderer.example.com:
  container_name: orderer.example.com
  image: hyperledger/fabric-orderer
  environment:
    - ORDERER_GENERAL_LOGLEVEL=debug
    - ORDERER_GENERAL_LISTENADDRESS=0.0.0.0
    - ORDERER_GENERAL_GENESISMETHOD=file
    - ORDERER_GENERAL_GENESISFILE=/var/hyperledger/orderer/orderer.genesis.block
    - ORDERER_GENERAL_LOCALMSPID=OrdererMSP
    - ORDERER_GENERAL_LOCALMSPDIR=/var/hyperledger/orderer/msp
    # 使能 TLS
```

```

- ORDERER_GENERAL_TLS_ENABLED=true
- ORDERER_GENERAL_TLS_PRIVATEKEY=/var/hyperledger/orderer/tls/
  server.key
- ORDERER_GENERAL_TLS_CERTIFICATE=/var/hyperledger/orderer/tls/
  server.crt
- ORDERER_GENERAL_TLS_ROOTCAS=[/var/hyperledger/orderer/tls/ca.crt]
working_dir: /opt/gopath/src/github.com/hyperledger/fabric
command: orderer
volumes:
- ../channel-artifacts/genesis.block:/var/hyperledger/orderer/
  orderer.genesis.block
- ../crypto-config/ordererOrganizations/example.com/orderers/orderer.
  example.com/msp:/var/hyperledger/orderer/msp
- ../crypto-config/ordererOrganizations/example.com/orderers/orderer.
  example.com/tls:/var/hyperledger/orderer/tls

ports:
- 7050:7050

peer0.org1.example.com:
  container_name: peer0.org1.example.com
  extends:
    file: peer-base.yaml
    service: peer-base
  environment:
    - CORE_PEER_ID=peer0.org1.example.com
    - CORE_PEER_ADDRESS=peer0.org1.example.com:7051
    - CORE_PEER_CHAINCODELISTENADDRESS=peer0.org1.example.com:7052
    - CORE_PEER_GOSSIP_EXTERNALENDPOINT=peer0.org1.example.com:7051
    - CORE_PEER_LOCALMSPID=Org1MSP
  volumes:
    - /var/run/:/host/var/run/
    - ../crypto-config/peerOrganizations/org1.example.com/peers/
      peer0.org1.example.com/msp:/etc/hyperledger/fabric/msp
    - ../crypto-config/peerOrganizations/org1.example.com/peers/
      peer0.org1.example.com/tls:/etc/hyperledger/fabric/tls
  ports:
    - 7051:7051
    - 7052:7052
    - 7053:7053

```

对于配置文件中环境变量和节点 YAML 之间的映射关系，请参考节的配置参数传递规则。从上面的配置文件可以看出，节点配置直接用到了 cryptogen 生成的目录结构文件。排序服务节点 orderer.example.com 的本地 MSP 名称是 OrdererMSP，本地 MSP 路径是 crypto-config/ordererOrganizations/example.com/orderers/orderer.example.com/msp，TLS 的 CA 根证书、密钥、证书文件目录是在 crypto-config/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls 下。Peer 节点 peer0.org1.example.com 的本地 MSP 名称是 Org1MSP，本地 MSP 路径是 crypto-config/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/msp，TLS 的 CA 根证书、密钥、证书文件目录是在 crypto-config/

peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls 下。

### 8.1.3 MSP 的配置最佳实践

本节将详尽地描述能够满足常用场合下的 MSP 配置最佳实践。

#### 1. 组织与 MSP 之间建立映射关系

我们建议实际的组织和 MSP 之间建立一一对应关系。也可以选择其他类型的映射关系，我们一起来看一下。

1) 一个组织对应多个 MSP 的情况。这种情况是一个组织有多个部门，从方便管理的角度或者隐私保护的角度而言，每个部门都要设置不同的 MSP。每个 Peer 节点只设置一个 MSP，同一组织内不同 MSP 的 Peer 节点之间不能互相认证，这样相同组织的不同部门之间不会同步数据。

2) 多个组织对应一个 MSP。这种情况是同一个联盟的不同组织之间采用相同的成员管理架构，数据会在不同组织之间同步。在前面的章节中我们已经了解到，在 Peer 节点之间的 Gossip 通信中，数据是在相同通道配置了相同 MSP 的 Peer 节点之间同步的。如果多个组织对应一个 MSP，则数据就不会限制在组织内部，会跨组织进行同步。

其实这是由 MSP 定义的粒度问题，一个 MSP 可以和一个组织对应，也可以和多个组织对应，还可以和一个组织内部的多个部门对应，根据 MSP 配置好 Peer 节点后，数据同步就限制在了 MSP 定义的范围内。

#### 2. 一个组织内部实现不同的权限控制

一个组织内部有多个部门，从而实现不同部门的权限控制。有两种方法可以实现这个场景：

##### (1) 给组织内的所有部门定义一个 MSP

给 Peer 节点配置 MSP 的时候，包含相同的可信根 CA 证书列表、中间 CA 证书、管理员证书，不同的 Peer 节点设置不同的所属部门。节点所属的部门是利用证书和部门之间映射的 `OrganizationalUnitIdentifiers` 定义的，它包含在 MSP 目录下的配置文件 “`config.yaml`” 中，本章前面已经介绍过。在第 3 章的相关部分，还介绍过可以按照基于部门验证的方法来定义交易背书策略和通道管理策略，这样就可以实现不同的权限控制了。这种方法会有一个问题，就是数据实际还是会在不同的 Peer 节点之间同步。因为 Peer 节点在识别组织身份类型 `OrgIdentityType` 的时候获取的是 MSP 标识，它会认为通道内相同 MSP 的节点都是可以分发数据的。

##### (2) 给组织内的每个部门单独定义 MSP

给 Peer 节点配置 MSP 的时候，不同部门配置的可信中间 CA 证书、管理员证书可以是不同的，不同部门成员的证书路径也是不同的。这种方式解决了所有部门定义在一个 MSP 中的问题，但是会带来管理上的复杂度。另外一个办法是每个部门都设置不同的 MSP，利



用证书和部门之间映射的 `OrganizationalUnitIdentifiers` 实现不同部门的权限控制，数据同步仍然会限制在组织的不同部门内，这同样也会有管理上的复杂度。

### 3. 不同类型的节点分别使用不同的 MSP

可能有这样的需求，希望给客户端、Peer 节点、排序服务节点分别设置不同的 MSP，因为身份信息会包含 MSP 标识，设置不同的 MSP 后就能确定身份类型。这在很多情况下是有用的，比如能够验证背书的确是由 Peer 节点签名的，而不是由客户端或者排序服务节点签名的。实际上，这样设置会带来一些问题，我们一起来分析一下。

不同节点类型分别设置不同的 MSP，对应的可信中间 CA 证书也不一样。在通道设置的时候需要包含不同的 MSP 及其可信中间 CA 证书，组织内的不同 MSP 成员才可以访问通道里的数据。同时背书策略可以指定只有 Peer 节点对应的 MSP 成员背书的交易才有效，这就能实现只能是 Peer 节点才能背书签名的目的。同一个组织按不同节点类型设置 MSP 之后，Peer 节点都有相同的 MSP，Peer 节点之间的数据同步不会受到影响，但会影响 Peer 节点和客户端之间的交互。

在 Peer 节点上某些系统链码的调用是和本地 MSP 相关的，比如只执行本地 MSP 配置的由管理员发起的安装链码（install）请求、加入通道（JoinChain）请求等。应用程序执行这类系统调用时还需要用到 Peer 节点相同 MSP 的管理员签名密钥和证书。当然，如果不把这类系统调用功能放在应用程序中去实现，正常的背书请求响应是没有问题的，因为 Peer 节点接收背书请求检查的是有没有通道的写入权限。因此可以在配置通道管理策略时增加客户端的 MSP，客户端就可以向 Peer 节点提交请求了。

还有一个问题是，注册事件回调函数的时候，Peer 节点只处理和本地 MSP 相同的客户端发起的请求，这个时候若 Peer 节点和客户端属于不同的 MSP，就会拒绝客户端发起的请求。如果业务依赖事件处理的话，应用程序和 Peer 节点还是需要采用相同的 MSP。

### 4. 区分管理员和 CA 证书

不要把可信根 CA 证书或者中间 CA 证书设置成 MSP 管理员证书，这样能把成员管理、签发证书与验证证书等不同职责拆分开来，方便管理和问题定位，这其实是一种常见的安全做法。

### 5. 区分根 CA 证书和 TLS 的根 CA 证书

MSP 的根 CA 证书和 TLS 根 CA 证书以及相关的中间 CA 证书需要存放在不同的文件夹中，这是为了避免混淆不同类别的证书。虽然并没有禁止根 CA 证书和 TLS 根 CA 证书采用相同的证书，在生产环境中建议还是区分开。

### 6. 吊销已经颁发的证书

由于权限管理或者其他方面的原因，已经颁发的证书是可以被吊销的。参考前面 MSP 成员证书的有效性验证过程，吊销已颁发的证书有多种办法：

- 1) 删除中间 CA 证书：删除 `intermediatecerts` 目录下的证书，这样由中间 CA 证书签



Fabric CA 服务端提供用户登记和注册的数字证书管理功能，数据存储后端可以是 MySQL、PostgreSQL、LDAP 等。如果配置了 LDAP，用户信息存在于 LDAP 中，而不是存放在 MySQL 或者 PostgreSQL 数据库中。通过数据存储和业务逻辑的分离，Fabric CA 服务能够采用无状态的集群部署，通过 HAProxy 等软件实现负载均衡功能，实现服务的高可用。Fabric CA 服务端提供了 RESTful 的接口供客户端工具和 HFC SDK 访问。手工部署的方式可以采用客户端工具来实现，如果集成到应用程序中，可以采用 HFC SDK 来实现。通过 HFC SDK 注册的证书有多种类型，包括 user、app、peer、orderer、client、validator、auditor 等，这些在 Peer 节点、Orderer 节点的部署时会用到，应用程序提交交易请求的时候也会用到。

## 8.2.2 Fabric CA 服务端的安装部署

本节介绍 Fabric CA 服务端的安装部署。

### (1) 准备工作

安装 Fabric CA 之前需要做一些准备工作：

- ❑ 安装 Go 语言 1.9 以上版本；
- ❑ 设置好 GOPATH 环境变量；
- ❑ 确认 libtool 和 libtdhl-dev 已安装。

Go 语言的安装请参考第 11 章的相关内容。在 Ubuntu 系统下可以直接通过命令行安装 libtool 和 libtdhl-dev：

```
sudo apt install libtool libtdhl-dev
```

### (2) 安装 Fabric CA 服务端和客户端

使用下面的命令即可将 fabric-ca-server 和 fabric-ca-client 安装至 \$GOPATH/bin 下：

```
go get -u github.com/hyperledger/fabric-ca/cmd/...
```

### (3) 通过命令行启动 Fabric CA 服务

初始化 fabric-ca-server：

```
fabric-ca-server init -b admin:adminpw
```

-b 选项提供注册用户的名称和密码，如果没有使用 LDAP，这个选项是必需的。默认的配置文件的名称为 fabric-ca-server-config.yaml，路径可以自定义。

启动 fabric-ca-server，使用默认设置：

```
fabric-ca-server start -b admin:adminpw
```

-b 选项提供注册用户的名称和密码，如果没有使用 LDAP，这个选项是必需的。默认的配置文件的名称为 fabric-ca-server-config.yaml，路径可以自定义。

#### (4) 通过 Docker Hub 下载的镜像启动 Fabric CA 服务

访问 <https://hub.docker.com/r/hyperledger/fabric-ca/tags>, 找到你想要获取的 fabric-ca 的版本。进入代码目录 \$GOPATH/src/github.com/hyperledger/fabric-ca/docker/server, 在编辑器中打开 docker-compose.yml, 修改 image 为指定的镜像版本。下面是这个文件的示例:

```
fabric-ca-server:
  image: hyperledger/fabric-ca:x86_64-1.0.0
  container_name: fabric-ca-server
  ports:
    - "7054:7054"
  environment:
    - FABRIC_CA_HOME=/etc/hyperledger/fabric-ca-server
  volumes:
    - "./fabric-ca-server:/etc/hyperledger/fabric-ca-server"
  command: sh -c 'fabric-ca-server start -b admin:adminpw'
```

打开一个终端, 进入 docker-compose.yml 所在的目录, 执行如下命令:

```
docker-compose up -d
```

如果指定的镜像不存在, 则 Docker 会主动拉取此镜像, 然后启动 fabric-ca 服务实例。

#### (5) 通过源码编译生成的 Docker 镜像, 以启动 Fabric CA 服务

你可以通过下面的命令使用 docker-compose 编译和启动服务:

```
cd $GOPATH/src/github.com/hyperledger/fabric-ca
make docker
cd docker/server
docker-compose up -d
```

hyperledger/fabric-ca 镜像同时包含 Fabric CA 的服务端 fabric-ca-server 和客户端 fabric-ca-client。

#### (6) fabric-ca-server 命令行选项

```
// 帮助:
    fabric-ca-server [command]
// 子命令:
    init                // 初始化 fabric-ca-server
    start               // 启动 fabric-ca-server
    version             // 打印 fabric-ca-server 版本信息
// 选项:
    --address string    // fabric-ca-server 的监听地址 (默认 "0.0.0.0")
    -b, --boot string   // 指定启动时的用户和密码, 格式为 user:pass
    --ca.certfile string // PEM 编码的证书文件 (默认 "ca-cert.pem")
    --ca.chainfile string // PEM 编码的链文件 (默认 "ca-chain.pem")
    --ca.keyfile string  // PEM 编码的密钥文件 (默认 "ca-key.pem")
    -n, --ca.name string // CA 名字
    --cacount int        // 非默认 CA 实例的个数
    --cafiles stringSlice // 逗号分隔的 CA 配置文件
```

```

--crl.expiry duration          // 通过 gencrl 请求生成 CRL 的过期时间 (默认 24h0m0s)
--crlsizelimit int            // 可接受的 CRL 大小限制 (默认 512000)
--csr.cn string               // 对上级 CA 的证书签名请求的 CN
--csr.hosts stringSlice       // 对上级 CA 的证书签名请求中的主机列表, 空格分割
--csr.serialnumber string     // 对上级 CA 的证书签名请求中的序列号
--db.datasource string        // 数据库特定的数据源 (默认 "fabric-ca-server.db")
--db.tls.certfiles stringSlice
// PEM 格式的信任的证书文件列表, 逗号分隔 (例如 . root1.pem,root2.pem)
--db.tls.client.certfile string
// DB 客户端 PEM 编码的证书文件, 仅当服务器要求双向认证时才指定
--db.tls.client.keyfile string
// DB 客户端 PEM 编码的密钥文件, 仅当服务器要求双向认证时才指定
--db.type string
// 数据库的类型, 可以是 SQLite、Postgres、MySQL (默认 sqlite3)
-d, --debug                  // 打开调试级别日志
-H, --home string
// server 的 home 目录 (默认 "/etc/hyperledger/fabric-ca")
--intermediate.enrollment.label string // 使用 HSM 操作的标签
--intermediate.enrollment.profile string // 发行证书时签名 profile 的名字
--intermediate.parentserver.caname string // 连接上级 CA 时用的 CN
-u, --intermediate.parentserver.url string
// 上级 CA 的 URL (格式 http://<username>:<password>@<address>:<port>)
--intermediate.tls.certfiles stringSlice
// 逗号分隔的 PEM 编码的证书列表 (e.g. root1.pem,root2.pem)
--intermediate.tls.client.certfile string
// PEM 编码的证书文件, 仅当服务器要求双向认证时才指定
--intermediate.tls.client.keyfile string
// 客户端 PEM 编码的密钥文件, 仅当服务器要求双向认证时才指定
--ldap.enabled                // 打开 LDAP
--ldap.groupfilter string
// LDAP 组过滤 (默认 "(memberUid=%s)")
--ldap.tls.certfiles stringSlice
// LDAP 信任的, 用逗号分隔的证书列表 (例如, root1.pem,root2.pem)
--ldap.tls.client.certfile string
// LDAP 客户端 PEM 编码的证书文件, 仅当服务器要求双向认证时才指定
--ldap.tls.client.keyfile string
// 客户端 PEM 编码的密钥文件, 仅当服务器要求双向认证时才指定
--ldap.url string
// LDAP 客户端 URL (格式 ldap:adminDN:adminPassword@host[:port]/base)
--ldap.userfilter string
// LDAP 用户过滤器 (默认 "(uid=%s)")
-p, --port int
// fabric-ca-server 监听端口 (默认 7054)
--registry.maxenrollments int
// 注册的最大次数, LDAP 关闭时有效 (默认 -1)
--tls.certfile string
// 服务器监听地址所用的 PEM 编码的 TLS 证书文件 (默认 "tls-cert.pem")
--tls.clientauth.certfiles stringSlice
// 逗号分隔的可信的客户端证书列表 (例如 . root1.pem,root2.pem)
--tls.clientauth.type string
// 客户端认证策略. (默认 "noclientcert")

```

```
--tls.enabled // 以 TLS 方式监听
--tls.keyfile string
// 服务器监听地址所用的 PEM 编码的私钥文件
```

使用“fabric-ca-server [command] --help”可获取更多子命令信息。

下面我们来看一下 Fabric CA 的数据库存储。默认的数据库采用的是嵌入式数据 SQLite，文件名是 fabric-ca-server.db。如果需要考虑集群部署，可以采用 MySQL 或者 PostgreSQL 数据库。

### 1. 基于 MySQL 的数据存储

在 Fabric CA 服务端的配置文件中添加下面的内容就可以访问 MySQL 数据库。但要确保其他相关变量也要配置正确。比如数据库名字对字符集的限制，可参考 MySQL 文档 <https://dev.mysql.com/doc/refman/5.7/en/identifiers.html>。

```
db:
  type: mysql
  datasource: root:rootpw@tcp(localhost:3306)/fabric_ca?parseTime=true&tls=
  custom
```

如果要通过 TLS 连接 MySQL 数据库，必须设置 db.tls，同时 MySQL 也要配置允许 TLS 访问。配置文件的示例如下所示：

```
db:
  ...
  tls:
    enabled: true
    certfiles:
      - db-server-cert.pem
    client:
      certfile: db-client-cert.pem
      keyfile: db-client-key.pem
```

其中，certfiles 是 PEM 编码的可信根证书文件列表，certfile 和 keyfile 是 PEM 编码的证书和密钥文件，用于 Fabric CA 服务器与 MySQL 数据库之间的安全连接。

### 2. 基于 PostgreSQL 的数据存储

如果采用 PostgreSQL 数据库，Fabric CA 服务端的配置文件可以参考如下的配置：

```
db:
  type: postgres
  datasource: host=localhost port=5432 user=Username password=Password
  dbname=fabric_ca sslmode=verify-full
```

在 PostgreSQL 上配置 SSL 的步骤如下：

- 1) 在 postgresql.conf 中，打开 SSL，设置为 on (SSL=on)。
- 2) 将你信任的 CA 证书 root.crt 放在 PostgreSQL 的 data 目录中。
- 3) 在 pg\_hba.conf 中 hostssl 位置，设置 clientcert 参数为 1。

关于如何生成签名证书，可参考 <https://www.postgresql.org/docs/9.5/static/ssl-tcp.html>，自签名证书可用于测试，不应该用于产品环境中。

更多详细配置，可参考 PostgreSQL 的官方文档 <https://www.postgresql.org/docs/9.4/static/libpq-ssl.html>。

### 3. 基于 LDAP 的数据存储

Fabric CA 服务端可以通过配置连接至 LDAP 服务器，实现如下的功能：

- 用户注册时从 LDAP 服务器中读取信息进行认证；
- 用户鉴权时从 LDAP 服务器中读取属性信息进行验证。

配置的方式是修改 Fabric CA 服务器配置文件内的 LDAP 选项，如下是配置文件的模板：

```
ldap:
  # 使能或者禁止 LDAP 客户端 ( 默认为 : false )
  enabled: false
  # LDAP 服务器的 URL
  url: <scheme>://<adminDN>:<adminPassword>@<host>:<port>/<base>
  userfilter: filter
```

其中：

- scheme 为 ldap 或 ldaps；
- adminDN 为管理员唯一的名字；
- pass 为管理员的密码；
- host LDAP 为服务器的主机名或 IP 地址；
- Port 为可选端口，默认值为 389 (ldap) 或 636 (ldaps)；
- base 为用于搜索 LDAP 树的根路径；
- filter 为过滤器，在搜索时将登录名转换为唯一名字。

下面是一个具体的示例，可连接默认配置的 OpenLDAP 服务器。

```
ldap:
  enabled: true
  url:
  ldap://cn=admin,dc=example,dc=org:admin@localhost:10389/dc=example,dc=org
  userfilter: (uid=%s)
```

OpenLDAP 服务的配置使用可参考 <https://github.com/osixia/docker-openldap> 上的说明。

配置好 LDAP 服务器后，用户注册过程如下所示：

- 1) Fabric CA 客户端或客户端 SDK 发送带有基本授权头部的用户注册请求；
- 2) Fabric CA 服务器接收到用户注册请求后，解析出头部中的用户名称和注册密码。

通过在配置文件中设置“userfilter”可以查找用户对应的可识别名称 (Distinguished Name, DN)，尝试执行 LDAP 绑定用户的注册密码进行身份验证。如果绑定成功，则用户注册就被认证通过了。



## 8.2.3 Fabric CA 服务端的操作使用

本节我们会介绍两种访问 Fabric CA 服务端的方法：Fabric CA 客户端工具和 RESTful 接口，客户端本身也是调用服务端 RESTful 接口实现的。客户端工具能比较直观地使用命令就能实现和服务端的交互，所以我们先介绍客户端工具，再介绍服务端的 RESTful 接口。

### 1. Fabric CA 客户端的使用

本节先介绍 fabric-ca-client 工具的命令行选项，再逐个介绍各个子命令的使用。

#### (1) fabric-ca-client 命令行选项

我们先看一下 Fabric CA 客户端的命令行选项：

```
// 帮助
fabric-ca-client [command]

// 子命令：
enroll      // 用户注册
gencrl      // 生成 CRL
gencsr      // 生成 CSR
getcacert   // 获取 CA 证书链
reenroll    // 重新注册
register     // 用户登记
revoke      // 用户注销
version     // 打印 Fabric CA 客户端版本

// 参数
--caname string           // CA 的名字
--csr.cn string           // 证书签名请求的 CN 名字
--csr.hosts stringSlice   // 证书签名请求的主机名列表，用空格分隔
--csr.names stringSlice   // 证书签名请求的名字列表，格式为 <name>=<value> (例如，C=CA,O=Org1)
--csr.serialnumber string // 证书签名请求的序列号
-d, --debug              // 打开调试级别日志
--enrollment.attrs stringSlice
    // 用逗号分隔的属性列表，格式为 <name>[:opt] (例如，foo,bar:opt)
--enrollment.label string // 使用 HSM 的标签
--enrollment.profile string // 发行证书使用签名 profile 的名字
-H, --home string         // 客户端 home 目录 (默认为 "$HOME/.fabric-ca-client")
--id.affiliation string    // 身份对应的隶属关系
--id.attrs stringSlice    // 用逗号分隔的属性列表，格式为 <name>=<value> (例如，foo=foo1,bar=bar1)
--id.maxenrollments int   // 注册的最大次数 (默认为 -1)
--id.name string          // 注册者身份的唯一名字 (DN)
--id.secret string        // 注册者的密码
--id.type string          // 注册者的类型 (例如，'peer', 'app', 'user') (默认为 "user")
-M, --mspdir string       // MSP 的目录路径 (默认为 "msp")
-m, --myhost string
```

```
// 注册时指定 CSR 中的主机名 (默认为 "$HOSTNAME")
-a, --revoke.aki string           // 注销证书的 AKI
-e, --revoke.name string          // 注销的身份名字
-r, --revoke.reason string        // 注销原因
-s, --revoke.serial string        // 注销证书的序列号
--tls.certfiles stringSlice
    // PEM 编码的信任证书列表, 用逗号分隔 (例如, root1.pem, root2.pem)
--tls.client.certfile string
    // 客户端 PEM 编码的证书, 仅当服务器要求双向认证时才指定
--tls.client.keyfile string
    // 客户端 PEM 编码的密钥, 仅当服务器要求双向认证时才指定
-u, --url string                  // fabric-ca-server 的 URL (默认为 "http://localhost:7054")
```

使用 “fabric-ca-client [command] --help” 可查看子命令帮助

后面我们会介绍其中主要的命令行功能。

## (2) 注册初始化的管理员用户

在 Fabric CA 服务端启动的时候有一个管理员用户, 需要先注册初始化的管理员用户, 获取注册证书以后才能进行后续的操作。注册初始化的管理员用户的步骤如下:

首先, 在 Fabric CA 服务端配置文件 fabric-ca-server-config.yaml 中设置客户端的 CSR (证书签名请求), 里面的选项是可以自定义的, 只有 “csr.cn” 必须设置成初始化的管理员 ID。默认的 CSR 配置如下:

```
csr:
  cn: <<enrollment ID>>
  key:
    algo: ecdsa
    size: 256
  names:
    - C: US
      ST: North Carolina
      L:
      O: Hyperledger Fabric
      OU: Fabric CA
  hosts:
    - <<hostname of the fabric-ca-client>>
  ca:
    pathlen:
    pathlenzero:
    expiry:
```

然后, 运行 fabric-ca-client enroll 命令来获取注册证书。在下面的命令中, 先设置获取注册证书存储目录的环境变量, 访问运行在本地 7054 端口的 fabric-ca 服务端, 注册的用户 ID 和密码分别是 admin 和 adminpw。

```
export FABRIC_CA_CLIENT_HOME=$HOME/fabric-ca/clients/admin
fabric-ca-client enroll -u http://admin:adminpw@localhost:7054
```

上面的注册命令将获取的注册证书 ECert 和对应的私钥, CA 证书链 PEM 文件存储在了环境变量 FABRIC\_CA\_CLIENT\_HOME 目录下的 msp 子目录中。

### (3) 登记一个新用户

只有已经注册的用户才可以发起登记 (Register) 请求, 发起登记请求的用户称为登记员 (Registrar), 登记新用户的时候还需要有相应的权限。Fabric CA 服务端在接收到登记请求时需要进行如下几个方面的检查:

1) 登记员需要有登记用户的登记权限: 登记员可以登记的用户类型记录在 “hf.Registrar.Roles” 属性中, 如果这个属性保存的内容是 “peer, app, user”, 则登记员就可以登记 peer、app 和 user 类型的用户, 但是不能登记 orderer 类型的用户。

2) 登记员只能登记自己归属范围内的用户: 比如登记员的归属是 a.b, 可以登记归属是 a.b.c 的用户, 不能登记归属是 a.c 的用户。如果不指定登记用户的归属, 那么默认就和登记员的归属一样。

3) 登记的用户属性需要满足如下一些条件。

① 登记的用户属性需要包含在登记员的用户属性 “hf.Registrar.Attributes” 中。目前只支持 “\*” 通配符, 比如 a.b.\* 代表所有以 a.b 开头的属性名称。

② 如果登记的用户也有 hf.Registrar.Attributes 属性, 需要其是登记员用户属性 hf.Registrar.Attributes 的子集。比如登记员的 hf.Registrar.Attributes 属性值是 a.b.\*, x.y.z, 则登记用户的 hf.Registrar.Attributes 属性值是 a.b.c, x.y.z 就是允许的, 因为登记用户的属性 a.b.c 满足登记员的属性 a.b.\*, 同时都有相同的属性 x.y.z。

我们再来看几个在登记用户时属性检查的几个例子, 表 8-6 所示为登记员和登记用户属性的名称都是 hf.Registrar.Attributes。

表 8-6 登记用户时的属性检查示例

登记员属性	登记用户的属性	是否可登记	登记说明
a.b.*, x.y.z	a.b.c	可以	登记用户的属性 “a.b.c” 满足登记员的属性 “a.b.*”
a.b.*, x.y.z	x.y.z	可以	登记员有和登记用户相同的属性 “x.y.z”
a.b.*, x.y.z	a.b.c, x.y.z	可以	登记用户的属性 “a.b.c” 满足登记员的属性 “a.b.*”, 同时都有相同的属性 “x.y.z”
a.b.*, x.y.z	a.b.c, x.y.*	不可以	登记用户的属性 “x.y.*” 范围大于登记员的属性 “x.y.z”
a.b.*, x.y.z	a.b.c, x.y.z, attr1	不可以	登记员的属性不包含 “attr1”
a.b.*, x.y.z	a.b	不可以	登记用户的属性 “a.b” 不包含在登记员的属性 “a.b.*” 中
a.b.*, x.y.z	x.y	不可以	登记用户的属性 “x.y” 不包含在登记员的属性 “x.y.z” 中

在下面的命令中, 登记员是 admin, 登记的新用户名称是 “admin2”, affiliation 属性为 “org1.department1”, 属性名 “hf.Revoker” 的值是 “true”, 属性名 “admin” 的值是 “true”。命令中 “:ecert” 后缀的意思是 admin 属性及其值会添加到用户注册证书中, 从而用来实现访问控制。

```
export FABRIC_CA_CLIENT_HOME=$HOME/fabric-ca/clients/admin
fabric-ca-client register --id.name admin2 --id.affiliation org1.department1 --id.
attrs 'hf.Revoker=true,admin=true:ecert'
```

命令运行后会打印注册密码，用户注册的时候需要这个注册密码。

在登记用户的时候可以使用“-id.attrs”选项同时指定多个属性，属性之间用逗号分隔，每个属性使用双引号，下面是一个具体的例子，同时登记了“hf.Registrar.Roles”属性和“hf.Revoker”属性：

```
fabric-ca-client register -d --id.name admin2 --id.affiliation org1.department1
--id.attrs '"hf.Registrar.Roles=peer,user",hf.Revoker=true'
```

指定属性的“-id.attrs”选项也可以有多个同时使用，下面的例子登记的是相同的“hf.Registrar.Roles”属性和“hf.Revoker”属性：

```
fabric-ca-client register -d --id.name admin2 --id.affiliation org1.department1
--id.attrs '"hf.Registrar.Roles=peer,user"' --id.attrs hf.Revoker=true
```

登记用户的默认值可以保存在 Fabric CA 的客户端配置文件中，假设配置文件如下所示：

```
id:
  name:
  type: user
  affiliation: org1.department1
  maxenrollments: -1
  attributes:
    - name: hf.Revoker
      value: true
    - name: anotherAttrName
      value: anotherAttrValue
```

用客户端命令行工具执行下面的命令登记用户，这样会从上面的配置文件中读取默认值，包括身份类型“user”，归属“org1.department1”，以及两个属性“hf.Revoker”和“anotherAttrName”。

```
export FABRIC_CA_CLIENT_HOME=$HOME/fabric-ca/clients/admin
fabric-ca-client register --id.name admin3
```

下面我们登记一个 Peer 节点，登记时指定用户类型为“peer”，用户名称是“peer1”，用户归属是“org1.department1”，注册密码不采用服务端自动生成，手动设置为“peer1pw”：

```
export FABRIC_CA_CLIENT_HOME=$HOME/fabric-ca/clients/admin
fabric-ca-client register --id.name peer1 --id.type peer --id.affiliation org1.
department1 --id.secret peer1pw
```

#### (4) Peer 节点的注册

通过命令行在 Fabric CA 服务端登记了 Peer 节点后，就可以通过登记的用户名称和

注册密码获取注册证书了。在下面的命令中，用户名称和注册密码分别是：“peer1”和“peer1pw”，选项“-M”的值为 Peer 节点的 MSP 目录，它保存获取到的注册证书等信息。

```
export FABRIC_CA_CLIENT_HOME=$HOME/fabric-ca/clients/peer1
fabric-ca-client enroll -u http://peer1:peer1pw@localhost:7054 -M $FABRIC_CA_
CLIENT_HOME/msp
```

Orderer 节点的注册过程是类似的，只需将对应的 MSP 目录路径改为你的 orderer 对应的 MSP 目录。

### (5) 重新获取用户的注册证书

假如你的注册证书将要过期或者私钥泄露，那么你可以调用 reenroll 命令重新申请新的注册证书。

```
export FABRIC_CA_CLIENT_HOME=$HOME/fabric-ca/clients/peer1
fabric-ca-client reenroll
```

### (6) 用户注销

一个身份或证书是可以被注销的。注销身份会注销其拥有的所有证书，也会同时阻止其申请新的证书。注销证书只是对单个证书进行无效处理。

下面的命令禁用了一个身份，并注销了与身份相关的所有的证书。注销后，此身份发起的任何请求都会被拒绝。

```
fabric-ca-client revoke -e <enrollment_id> -r <reason>
```

例如，下面的命令使用 admin 注销了 peer1 的身份。

```
export FABRIC_CA_CLIENT_HOME=$HOME/fabric-ca/clients/admin
fabric-ca-client revoke -e peer1
```

也可以通过 API 和序列号注销：

```
fabric-ca-client revoke -a xxx -s yyy -r <reason>
```

例如，你可以使用 openssl 命令获得证书的 API 和序列号，然后传递给 revoke 命令

```
serial=$(openssl x509 -in userecert.pem -serial -noout | cut -d "=" -f 2)
aki=$(openssl x509 -in userecert.pem -text | awk '/keyid/ {gsub(/
*keyid:|:/, "", $1); print tolower($0)}')
fabric-ca-client revoke -s $serial -a $aki -r affiliationchange
```

### (7) 获取 CA 证书链

在 MSP 目录的子目录 cacerts 下，存储的是可信的 CA 根证书。可以通过 fabric-ca-client getcacerts 命令从 Fabric CA 服务器获取根 CA 证书。

我们来看一个例子，下面的命令启动一个名称为“CA2”的 Fabric CA 服务，监听的端口是 7055，初始的管理员用户名称和密码分别是：“admin”和“ca2pw”：

```
export FABRIC_CA_SERVER_HOME=$HOME/ca2
fabric-ca-server start -b admin:ca2pw -p 7055 -n CA2
```

下面的命令可以获取 CA2 的根 CA 证书并保存到 peer1 的 MSP 目录下：

```
export FABRIC_CA_CLIENT_HOME=$HOME/fabric-ca/clients/peer1
fabric-ca-client getcacert -u http://localhost:7055 -M $FABRIC_CA_CLIENT_HOME/
msp
```

## (8) 使用 TLS

Fabric CA 客户端可以通过配置 fabric-ca-client-config.yaml 使用 TLS。

```
tls:
# 使能 TLS (默认: false)
  enabled: true
  certfiles:
    - root.pem
  client:
    certfile: tls_client-cert.pem
    keyfile: tls_client-key.pem
```

其中，certfiles 选项设置信任的根证书，client 选项仅在服务器设置双向认证后才必须指定。

## 2. Fabric CA 的 RESTful 接口

Fabric CA 提供了多个接口，包括获取 CA 信息、获取注册证书、重新获取注册证书、用户登记、用户注销、批量获取交易证书等。Fabric CA 提供的 RESTful 的接口，可以通过 http 或 https 访问。

### (1) 获取 CA 信息

表 8-7 是获取 CA 信息请求的 URL。

获取 CA 信息请求的 Header 信息如下所示：

```
Content-Type: application/json
```

获取 CA 信息请求的 Body 内容如下所示：

```
{
  "caname": "ca.fabric.chainnova.xyz"
}
```

获取 CA 信息返回的 Body 如下所示：

```
{
  "success": true,
  "result": {
    "CAName": "ca.fabric.chainnova.xyz",
    "CAChain": "LS0tLS1CRUdJTlBDRVJUSUZJQ0FURS0tLS0tCk1JSUNVekNDQWZtZ0F3SUJBZ01RRnRlXz1piZy90UjZKUWw2SDR3TS9TakFLQmdncWhrak9QUVFEQWpCN01Rc3cKQ1FZRFZR"
  }
}
```

表 8-7 获取 CA 信息请求的 URL

请求类型	请求 URL
GET	/api/v1/cainfo

```
UUDFd0pWVXpFVE1CRUdBMVVFQ0JNS1EyRnNhV1p2Y201cF1URVdNQ1FHQTFVRUJ4TU5VMkZ1
SUVaeQpZVzVqYVhOamJ6RWRNQNnHQTFVRUNoTVVabUZpY21sakxtTm9ZV2x1Ym05M1lTNTR1
WG94SURBZUJnTlZCQU1UCKYyTmhMbVpoWW5KcF15NWphR0ZwYm01dmRtRXVlSGw2TUI0WERU
RTNNRGd5T1RBeU16Z3dNMW9YRFRJm01EZ3kKtNpBeU16Z3dNMW93ZXpFTE1Ba0dBmVVFQmhn
Q1ZWtXhFekFSQmdOVkZJBZ1RDa05oYkdsbWVzSnVhV0V4RmpBVQpCZ05WQkFjVERWTmhiaUJH
Y21GdVkybHpzMjh4SFRBYkJnTlZCQW9URkdaaFluSnBZeTVqYUdGcGJtNXZkbUV1CmVlY2ZlZDZ
U0F3SGdZRFZRUURFeGRqWVM1bVlXSnlhV011WTJoaGFXXVIM1poTG5oNWVqQ1pNQk1HQNlx
R1NNNDkKQWdFR0NDcUdTtTQ5QXdfSEEWsUFCT2tBVnhlaWZ4c09kSzU5enJmeXFNWEx1eFU3
MTI0SW1vdDRSYzN0VWpLNgpUVGt6dGRpNFJyUitXVDBoaHNzRVA3N1RBZjNGSVBzWXFoanZx
bXVzVUFxalh6QmRNQTRHQTFVZER3RUIvd1FFCkF3SUJwakFQmQOVkhTVUVDREHFQmdSVkht
VUFNQThHQTFVZEV3RUIvd1FGTUFNQkFmOHdLUVlEVl1IwT0JDSUUKSU1nWTDaOGx3OS9QdkSy
Wlk3T2tss1MzM3pKUU5wNjZzTWc5RTcwR01JL0VNQW9HQ0N0R1NNND1CQU1DQTBnQpNRVVD
SVFEZU5Qa1Z6UuZxbWlSUEJodDZyRmNCS3BDWnhPYVpjRkdzbmM3QW81Nkln01nRhOzSldz
TTFWRmozCnlxTVGxYXNzb1hLMU1FODhuVzBlVE5rUHDHTStsSxc9Ci0tLS0tRU5EIENFU1RJ
RklDQVRFLS0tLS0K"
},
"errors":{
},
"messages":{
}
}
```

(2) 获取注册证书

表 8-8 是用户注册请求的类型和 URL。

用户注册请求的 Header 信息如下所示：

```
Authorization: YWRtaW46YWRtaW5wdw==
Content-Type: application/json
```

用户注册请求的 Body 内容如下所示：

```
{
  "request": "string",
  "profile": "Unknown Type: string,null",
  "label": "Unknown Type: string,null",
  "caname": "Unknown Type: string,null"
}
```

用户注册请求返回的 Body 内容如下所示：

```
{
  "Success": true,
  "Result": "string",
  "Errors": [
    {
      "code": 0,
      "message": "string"
    }
  ],
  "Messages": [
```

表 8-8 用户注册请求的类型和 URL

用户注册请求的类型	用户注册请求的 URL
POST	/api/v1/enroll



```
{
  "code": 0,
  "message": "string"
}
```

(3) 重新获取注册证书

表 8-9 是用户重新注册请求的类型和 URL。

重新注册请求的 Header 信息如下所示：

Authorization: YWRtaW46YWRtaW5wdw==  
Content-Type: application/json

重新注册请求的 Body 内容如下所示：

```
{
  "request": "string",
  "profile": "Unknown Type: string,null",
  "label": "Unknown Type: string,null",
  "caname": "Unknown Type: string,null"
}
```

重新注册请求返回的 Body 内容如下所示：

```
{
  "Success": true,
  "Result": "string",
  "Errors": [
    {
      "code": 0,
      "message": "string"
    }
  ],
  "Messages": [
    {
      "code": 0,
      "message": "string"
    }
  ]
}
```

表 8-9 重新注册请求的类型和 URL

重新注册请求的类型	重新注册请求的 URL
POST	/api/v1/reenroll

(4) 用户登记

表 8-10 是用户登记请求的类型和 URL。

用户登记请求的 Header 信息如下所示：

Authorization: YWRtaW46YWRtaW5wdw==  
Content-Type: application/json

用户登记请求的 Body 内容如下所示：

表 8-10 用户登记请求的类型和 URL

用户登记请求的类型	用户登记请求的 URL
POST	/api/v1/register

```
{
  "id": "string",
  "type": "string",
  "secret": "Unknown Type: string,null",
  "max_enrollments": "Unknown Type: integer,null",
  "affiliation_path": "string",
  "attrs": [
    {
      "name": "string",
      "value": "string"
    }
  ],
  "caname": "Unknown Type: string,null"
}
```

用户登记请求返回的 Body 内容如下所示：

```
{
  "Success": true,
  "Result": {
    "credentials": "string"
  },
  "Errors": [
    {
      "code": 0,
      "message": "string"
    }
  ],
  "Messages": [
    {
      "code": 0,
      "message": "string"
    }
  ]
}
```

(5) 用户注销

表 8-11 是用户注销请求的类型和 URL。

表 8-11 用户注销的类型和 URL

注销请求	注销一个证书或某一身份所有的证书
POST	/api/v1/ revoke

用户注销请求的 Header 信息如下所示：

Authorization: YWRtaW46YWRtaW5wdw==  
Content-Type: application/json

用户注销请求的 Body 内容如下所示：

```
{
  "id": "Unknown Type: string,null",
  "aki": "Unknown Type: string,null",
  "serial": "Unknown Type: string,null",
  "reason": "Unknown Type: string,null",
  "caname": "Unknown Type: string,null"
}
```

用户注销请求返回的 Body 内容如下所示:

```
{
  "Success": true,
  "Result": "string",
  "Errors": [
    {
      "code": 0,
      "message": "string"
    }
  ],
  "Messages": [
    {
      "code": 0,
      "message": "string"
    }
  ]
}
```

#### (6) 批量获取交易证书

表 8-12 是批量获取交易证书请求的类型和 URL。

表 8-12 批量获取交易证书请求的类型和 URL

交易证书请求	批量获取的交易证书
POST	/api/v1/tcert

批量获取交易证书请求的 Header 信息如下所示:

```
Authorization: YWRtaW46YWRtaW5wdw==
Content-Type: application/json
```

批量获取交易证书请求的 Body 内容如下所示:

```
{
  "count": 0,
  "attr_names": [
    "string"
  ],
  "encrypt_attrs": true,
  "validity_period": 0,
  "caname": "Unknown Type: string,null"
}
```

批量获取交易证书请求返回的 Body 内容如下所示：

```
{
  "Success": true,
  "Result": {
    "id": 0,
    "ts": 0,
    "key": "string",
    "tcerts": [
      {
        "cert": "string",
        "keys": [
          {
            "name": "string",
            "value": "string"
          }
        ]
      }
    ]
  },
  "Errors": [
    {
      "code": 0,
      "message": "string"
    }
  ],
  "Messages": [
    {
      "code": 0,
      "message": "string"
    }
  ]
}
```

8.3 本章小结

本章介绍了与成员管理相关的两个服务，MSP 和 Fabric CA。Fabric CA 负责颁发证书，可以有多种方式实现。Fabric CA 是超级账本提供的默认实现，并不是强制使用的。MSP 利用生成的证书进行签名和验证，把不同的证书归属到不同的 MSP 中，利用 PKI 机制验证成员身份，实现成员的认证和授权管理。

## 支持多种语言的智能合约

智能合约 (smart contract) 最早是 1996 年尼克·萨博 (Nick Szabo) 在他的文章 “Smart Contracts: Building Blocks for Digital Markets” 中提出来的:

A smart contract is a set of promises, specified in digital form, including protocols within which the parties perform on these promises.

维基百科对智能合约做了扩展:

A smart contract is a computer protocol intended to facilitate, verify, or enforce the negotiation or performance of a contract. Smart contracts were first proposed by Nick Szabo in 1996.

Proponents of smart contracts claim that many kinds of contractual clauses may be made partially or fully self-executing, self-enforcing, or both. The aim with smart contracts is to provide security that is superior to traditional contract law and to reduce other transaction costs associated with contracting.

概括起来, 智能合约有如下几个特性。

1) 智能合约必须是一种合约。合约是平等的当事人之间执行约定内容的协议。参与方对合约的内容要达成一致, 并且遵守合约执行的结果。相比于传统合约, 不同的地方在于: 智能合约是“数字形式”的, 需要转换成计算机可读和可执行的代码。

2) 智能合约是部分或者完全自我执行 (self-executing) 和自我强制 (self-enforcing) 的: 自我执行是指合约能立即自动生效。自我强制是指合约的参与方能够根据最优的结果自行决定是否参与或者终止和关联方的关系, 继续参与得不偿失才会终止合约。整个决策过程是不需要可信的第三方干预的, 这是一种有用的特性, 因为第三方参与仲裁会有一些额外的开销。这是一种防欺诈的协议, 参与方会计算违约的得失, 做出理性的选择, 客观上避免了任何一个参与方欺骗对方。

3) 智能合约需要安全的运行环境: 智能合约的运行由相关方执行或者调用执行完成约定的业务逻辑, 运行环境是安全可靠的, 执行的结果能在各方达成一致。

4) 只有智能合约才能修改账本数据: 账本数据是各个节点各自独立维护的分布式数据, 智能合约执行的结果达成一致后才能修改账本数据, 这里的修改是指追加数据, 数据本身是不可篡改的。

本章介绍 Hyperledger Fabric 1.0 的智能合约, 即链码 (Chaincode)。

## 9.1 概述

链码是独立可运行的应用程序, 运行在基于 Docker 的安全容器中, 在启动的时候和背书节点建立 gRPC 连接, 在运行过程中通过接口和背书节点通信。链码是线下开发好再部署到链上的, 部署的时候需要一定的权限, 成功部署就代表了链码的业务逻辑是关联方已经达成一致的。达成一致的过程其实是在线下完成的, 线上的共识过程只是权限检查和业务逻辑的执行, 并没有线上审核的流程。也就是说, 如果链码并没有真正地做线下的确认, 只要有部署的权限, 也是可以直接上线的。目前智能合约部署的权限管理完全是基于成员管理的机制, 这种权限管理的粒度是很粗的, 不能做到完全像 RBAC 一样精确到某个资源。成员管理的内容参考前面相关的章节。

## 9.2 链码的生命周期管理

Hyperledger Fabric 1.0 中, 区块链网络中的各种节点 (Peer 节点、排序服务节点等) 都提供了 gRPC 接口, 只要有权限, 应用程序就可以访问它们提供的功能。背书节点是 Peer 节点的一种角色, 管理和维护了链上的链码, 可以通过背书节点开放的接口, 执行智能合约的功能。有两种方式可以访问接口, 命令行和各种语言的 SDK 都可以。前面的章节已经介绍过命令的方式, 第 10 章会详细介绍 SDK 的接口。

### 9.2.1 链码的生命周期

目前的版本中, 链码提供了 4 个管理链码生命周期的命令, 分别是链码的打包 (package)、安装 (install)、实例化 (instantiate)、升级 (upgrade)。在以后的版本中, 还可能提供链码的停止 (stop) 和启动 (start) 命令, 在不删除链码的情况下可以停止和启用链码。链码在安装和实例化以后, 就处于激活的状态, 应用程序或者命令行就可以调用和触发智能合约的功能了。链码安装以后随时都是可以升级的。

#### 1. 链码的打包

链码的内容主要包含以下 3 个部分。

□ 链码源码, 但需通过 ChaincodeDeploymentSpec 或 CDS 定义。CDS 依据代码及其他

一些属性（名称、版本等）来定义链码。

❑ 实例化策略，这是可选的，背书策略前面已经介绍过，可以参考前面相关的内容。

❑ 链码的签名。

链码的签名实现了下面三个目标：

❑ 表明是谁创建的链码；

❑ 允许验证链码包里的内容；

❑ 可以检测链码包是否被篡改。

链码的实例化策略会验证链码所有者的身份，进而验证其提交的链码源码、实例化策略是否有效。

### （1）链码的创建

创建链码有两种方式。

❑ 多个所有者：需要多个所有者对链码进行签名，先创建一个链码包 SignedChaincode DeploymentSpec，然后发送给多个所有者进行签名。

❑ 单一所有者：只有安装链码的节点对链码签名。

我们先来看复杂的情况。当然，如不需要了解多用户的情况，可以直接跳转到本节后面的“链码的安装”小节。

可通过下面的命令行创建带签名的链码包。

```
peer chaincode package -n mycc -p github.com/hyperledger/fabric/examples/chaincode/go/chaincode_example02 -v 0 -s -S -i "AND('OrgA.admin')" ccpack.out
```

其中，-s 选项生成一个有多个所有者签名的链码包，而不是简单地创建一个不带签名的 ChaincodeDeploymentSpec。如果指定了 -s 选项，当其他所有者要签名时，还需要指定 -S 选项。否则，创建的链码包 SignedChaincodeDeploymentSpec 只会在 ChaincodeDeploymentSpec 基础上添加实例化策略，不会包含所有者的签名。

-S 选项可以使 MSP (core.yaml 中 localMspid 属性值定义的) 对程序包进行签名。-S 是可选的，如果创建了一个没有签名的包，那么其他的所有者不能通过对其使用 signpackage 命令签名。

可以通过 -i 选项为链码指定实例化策略。实例化策略与背书策略类似，它指明哪些身份可以对链码实例化。在上面的例子中，只允许 OrgA 管理员进行链码实例化。如果没有提供任何策略，系统将会采用默认策略，该策略只允许 Peer 节点 MSP 的管理员实例化链码。

### （2）链码的签名

链码在创建时签了名才可以由其他的所有者校验签名和继续签名，签名的过程可以是线下操作 (out-of-band) 的。

链码 SignedChaincodeDeploymentSpec 是封装了 ChaincodeDeploymentSpec 的结构，主要是增加了实例化策略和所有者的签名，定义如下。

```
type SignedChaincodeDeploymentSpec struct {
```



```
// ChaincodeDeploymentSpec 序列号后的字节数组
ChaincodeDeploymentSpec []byte
// 实例化策略，结构同背书策略
InstantiationPolicy []byte
// 所有者的签名，可以是多个
OwnerEndorsements []*Endorsement
}
```

从上面的结构可以看出，主要包含如下几个部分。

❑ ChaincodeDeploymentSpec 包含链码的源码、名称和版本。

❑ 实例化策略，在实例化的时候 VSCC 会验证。

❑ 所有者的签名背书，包含了链码所有者的列表。

实例化策略定义了签名的 MSPPrincipal 和需要满足的规则，是需要线下确定的，链码实例化的时候会读取这些信息进行验证。如果不指定实例化策略，默认通道的管理员才能对链码实例化。实例化策略的定义如下。

```
type SignaturePolicyEnvelope struct {
    // 策略的版本
    Version    int32                `json:"version,omitempty"`
    // 策略的规则
    Rule       *SignaturePolicy        `json:"rule,omitempty"`
    // 策略的主体：可以是基于角色、基于组织和具体的身份
    Identities []*common1.MSPPrincipal `json:"identities,omitempty"`
}
```

链码所有者可以对一个之前创建好的链码包进行签名，具体使用如下命令。

```
peer chaincode signpackage ccpack.out signedccpack.out
```

其中，命令中的 ccpack.out 和 signedccpack.out 分别指输入与输出包。signedccpack.out 包含一个用本地 MSP 对包进行的附加签名。

## 2. 链码的安装

链码安装的时候会把链码的源码打包到前面已经介绍过的结构 Chaincode DeploymentSpec 中，安装到需要运行链码的 Peer 节点上。链码安装是针对节点的，每次安装只对单个节点有效。需要给每个要运行链码的背书节点都安装一遍，具体需要安装到哪些节点，可以根据背书策略来选择。

从安全角度考虑，链码应该只安装在需要执行的背书节点上，可以保护链码的逻辑被未授权的节点获取到。没有安装链码的节点是不能执行链码的智能合约的，但是可以验证链上的交易并且提交到本地账本中。

安装链码的命令如下。

```
peer chaincode install -n ChaincodeName -v version -p ChaincodePath
```

上述的 -n 选项标识的是链码的名字，-v 选项标识的是链码的版本，-p 选项标识的是链码源码的路径，其必须在用户设置的环境变量 GOPATH 目录下（比如 \$GOPATH/src/cc）。

链码安装的时候是需要管理员权限的，SignedProposal 必须是 Peer 节点本地 MSP 设置的一个管理员签名的。

### 3. 链码的实例化

链码的实例化会调用链码生命周期系统链码（LSCC）在通道上创建和初始化链码。链码在实例化之前是和通道无关的，实例化的时候才和通道绑定。链码可以和多个通道绑定，在通道上初始化后记录到通道的状态数据库中。同一个链码在不同通道上的数据是隔离的，不同通道之间不会有影响，这在前面的第7章已经介绍过。

链码实例化的时候会检查是否符合链码实例化的策略和通道的写入策略。实例化的策略验证过程是检查实例化交易的签名是否符合策略的规则，验证通过才会写入到账本和状态数据中。通道的写入策略是在创建通道时指定的，这也是从安全角度考虑的，避免未授权的用户部署链码和调用链码。前面我们提到过，默认的实例化策略是通道的任何一个管理员，所以链码的实例化交易提交者必须也是通道的管理员。

链码的实例化会指定链码的背书策略，用来确定通道上哪些节点执行的交易才能添加到账本中，这在前面的章节中已经介绍过。

命令行的链码实例化如下。

```
peer chaincode instantiate -n ChaincodeName -v version -c '{"Args":["john","0"]}'  
-P "OR ('Org1.member','Org2.member')"
```

上面的 -P 选项指定的就是背书策略，例子中的策略只要是 Org1 或者 Org2 的任意一个成员的背书就可以了。

目前命令行的链码实例化只能指定 AND 和 OR 的背书策略，内部会转换成 NOutOf 策略，这在 SDK 的接口中是可以指定的。

链码实例化以后，就处于 ready 状态了，可以处理链码的调用和查询交易了，本章后面部分会详细地介绍背书节点侧和链码侧的有限状态机。

### 4. 链码的升级

链码安装以后是随时可以升级的，链码的名称需要保持不变，必须要更新的是链码的版本，其他的部分，比如链码的所有者和实例化策略都是可选的。链码的版本是用字符串表示的，并没有指定比较版本大小的规则，是以更新的先后顺序为准的，具体的操作方法线下自行确定。

同样，升级之前还需要把链码安装到对应的背书节点上。链码的升级和实例化是类似的操作，也是绑定链码到通道上，执行初始化的操作。链码升级只会影响到指定的通道，没有绑定链码新版本的通道还是继续旧的版本。同一个背书节点上，不同通道绑定的链码可能是不同的版本，所以升级的时候不会主动删除旧的版本。

链码升级的时候同样会检查实例化策略，采用的是通道上链码最新版本的实例化策略，检查通过以后才会更新为链码新版本指定的实例化策略。

## 5. 链码的停止和启动

链码的停止与启动功能还没实现，只能手动删除链码的容器和镜像，再删除背书节点本地保存的链码。链码容器和镜像的命名规则可以参考本章后面的容器管理部分。链码代码默认是在下面的目录结构中。

```
/var/hyperledger/production/chaincodes/<ccname>:<ccversion>
```

其中，/var/hyperledger/production 目录是背书节点的环境变量 CORE\_PEER\_FILESYSTEMPATH 指定的。

## 9.2.2 应用程序和链码的交互流程

普通链码用来执行特定的智能合约功能，图 9-1 是应用程序和链码的交互流程图，包含如下几个步骤。

第 1 步：应用程序或者命令行通过 gRPC 请求向背书节点发起链码的调用请求，背书节点再转发给链码执行，应用程序不能直接和链码通信。

第 2 步：背书节点会检查对应的链码是否启动，检查的方法是查看本地维护的映射表里是否有指定链上的链码名称和版本的记录。如果没有相关的记录，就会通过 Docker 的 API 发起创建或者启动容器的命令。

第 3 步：Docker 服务根据 API 的命令启动容器，并建立和背书节点的 gRPC 连接。如果背书节点接收到请求以后检查链码容器已经启动，会直接跳过第 2 步和 3 步。

第 4 步：通过链码和背书节点建立好的 gRPC 连接，转发应用程序调用的请求，链码在执行过程中会和背书节点有多次的交互。

第 5 步：链码执行完以后，调用背书节点的 ESCC 对模拟执行的结果进行背书。

第 6 步：ESCC 对模拟执行进行签名，返回背书的结果。

第 7 步：背书节点返回包含背书节点背书的结果给应用程序。

在整个的交易流程中，链码只参与业务逻辑模拟执行的过程，后续的交易排序和验证分别是排序服务和记账节点完成的。背书节点接收和处理请求后就返回应用程序了，不会转发请求给其他背书节点。应用程序可以自由选择背书节点发起请求，只要最终生成的交

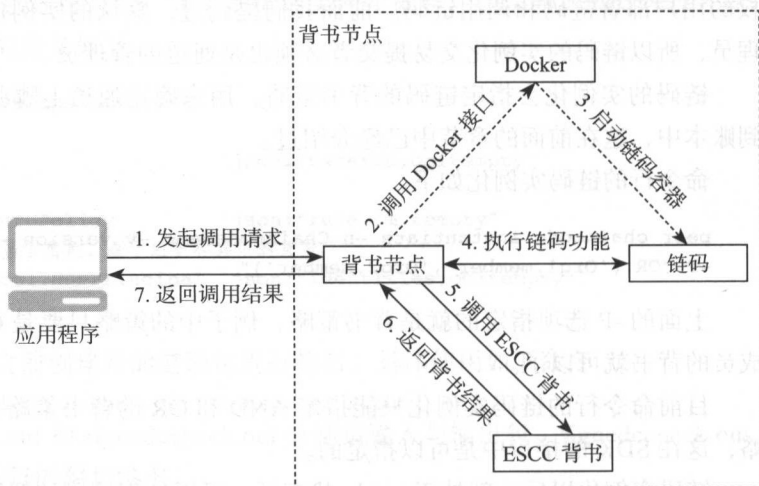


图 9-1 应用程序和链码的交互流程

易能够满足背书策略就可以。

### 9.2.3 背书节点接收应用程序的请求处理

应用程序通过 gRPC 的接口发起请求，命令如下。

```
ProcessProposal(ctx context.Context, in *SignedProposal, opts ...grpc.CallOption)
(*ProposalResponse, error)
```

背书节点接收到请求以后，会做一些必要的检查，比如是否有权限提交交易、是否是重复交易等，真正的执行过程是在链码中完成的，ESCC 最后对执行的结果进行签名背书。中间有任何异常都会终止后续的执行，返回结果给应用程序。链码调用时序图如图 9-2 所示。

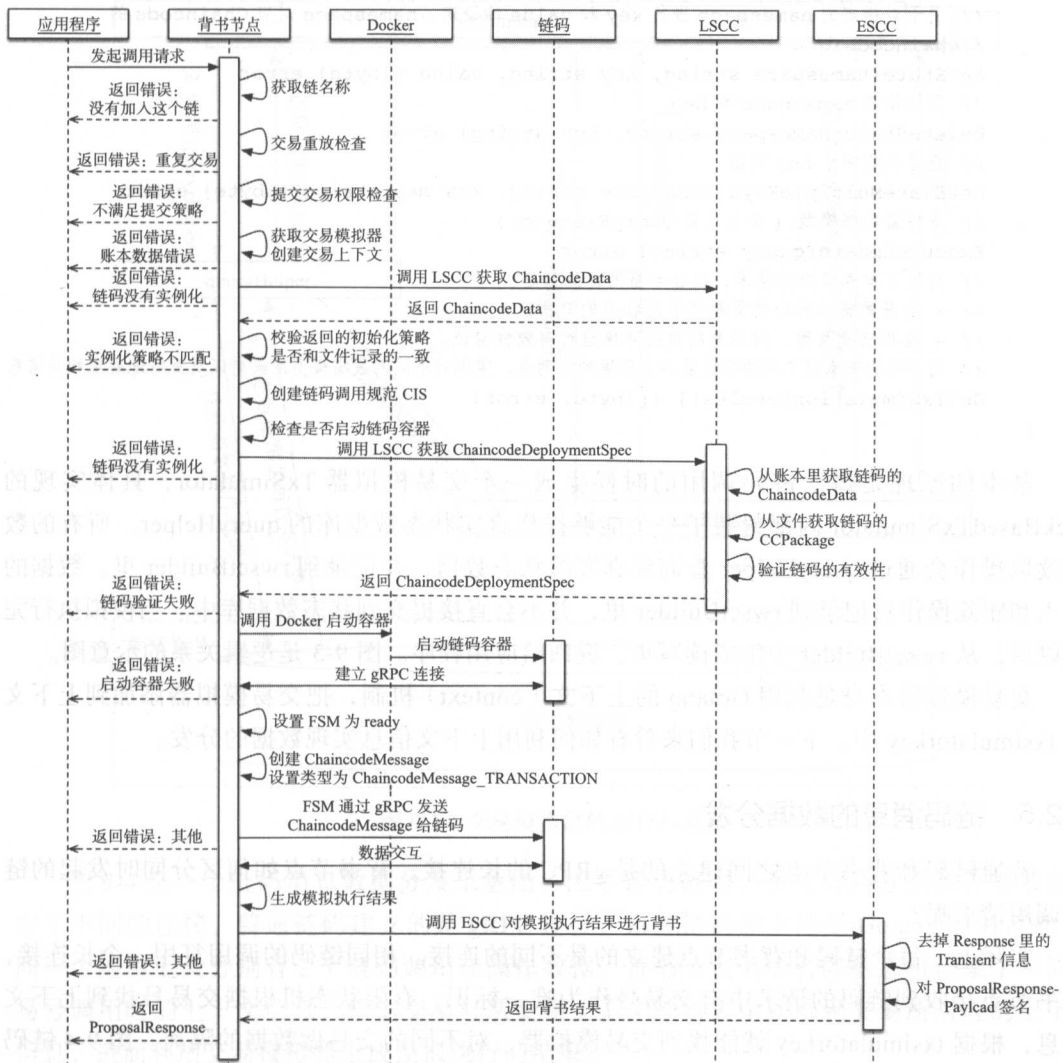


图 9-2 链码调用时序图

下面再来看几个重要流程的实现方式。

### 9.2.4 采用上下文实现交易的模拟执行

链码在业务逻辑的处理过程中会读取和操作账本数据，对这些数据的修改并没有直接影响到状态数据库，而是内部实现了一个交易模拟器，把过程数据记录到了模拟器中。交易模拟器的接口如下。

```
// TxSimulator 可以通过最新连续快照来模拟交易
// Set* 方法用于支持 KV-based 数据模型。ExecuteUpdate 方法用于支持富数据模型及相关查询
type TxSimulator interface {
    QueryExecutor
    // 用于对指定的 namespace 进行 key 和 value 的设定。namespace 对应 Chaincode 的
    // chaincodeId
    SetState(namespace string, key string, value []byte) error
    // 删除指定 namespace 和 key
    DeleteState(namespace string, key string) error
    // 批量设置多个 key 的值
    SetStateMultipleKeys(namespace string, kvs map[string][]byte) error
    // 支持富数据模型 (参见上面 QueryExecutor)
    ExecuteUpdate(query string) error
    // 封装了事务模拟的结果，包含丰富的详细信息：
    // - 交易的提交状态的更新将引起状态的更新；
    // - 在提交交易时，对交易的执行环境进行有效性验证。
    // 对不同的账本以不同的形式展示上面提到的两点，实现对不同的数据模型的支持以及更好地展现相关信息
    GetTxSimulationResults() ([]byte, error)
}
```

基本的原理是每次链码调用的时候生成一个交易模拟器 TxSimulator，具体实现的 lockBasedTxSimulator 内部封装了一个能够操作真实状态数据库的 queryHelper。所有的数据读取操作会通过 queryHelper 查询到真实的状态数据，并记录到 rwsetBuilder 里，数据的写入和删除操作只记录到 rwsetBuilder 里，并不会直接提交到状态数据库中。当模拟执行完成以后，从 rwsetBuilder 中生成读写集，返回给应用程序。图 9-3 是逻辑关系的示意图。

交易模拟器本身是利用 Golang 的上下文 (context) 机制，把交易模拟器添加到上下文的 txsimulatorkey 中。下一节我们来看看如何利用上下文信息实现数据的分发。

### 9.2.5 链码消息的数据分发

普通链码和背书节点之间建立的是 gRPC 的长连接，背书节点如何区分同时发起的链码调用请求呢？

实际上，每个链码和背书节点建立的是不同的连接。相同链码的调用复用同一个长连接，背书节点接收到链码的请求中有交易号作为唯一标识，有限状态机根据交易号找到上下文信息，根据 txsimulatorkey 就能找到交易模拟器，对不同的交易做数据的隔离。图 9-4 链码消息的数据分发示意图。

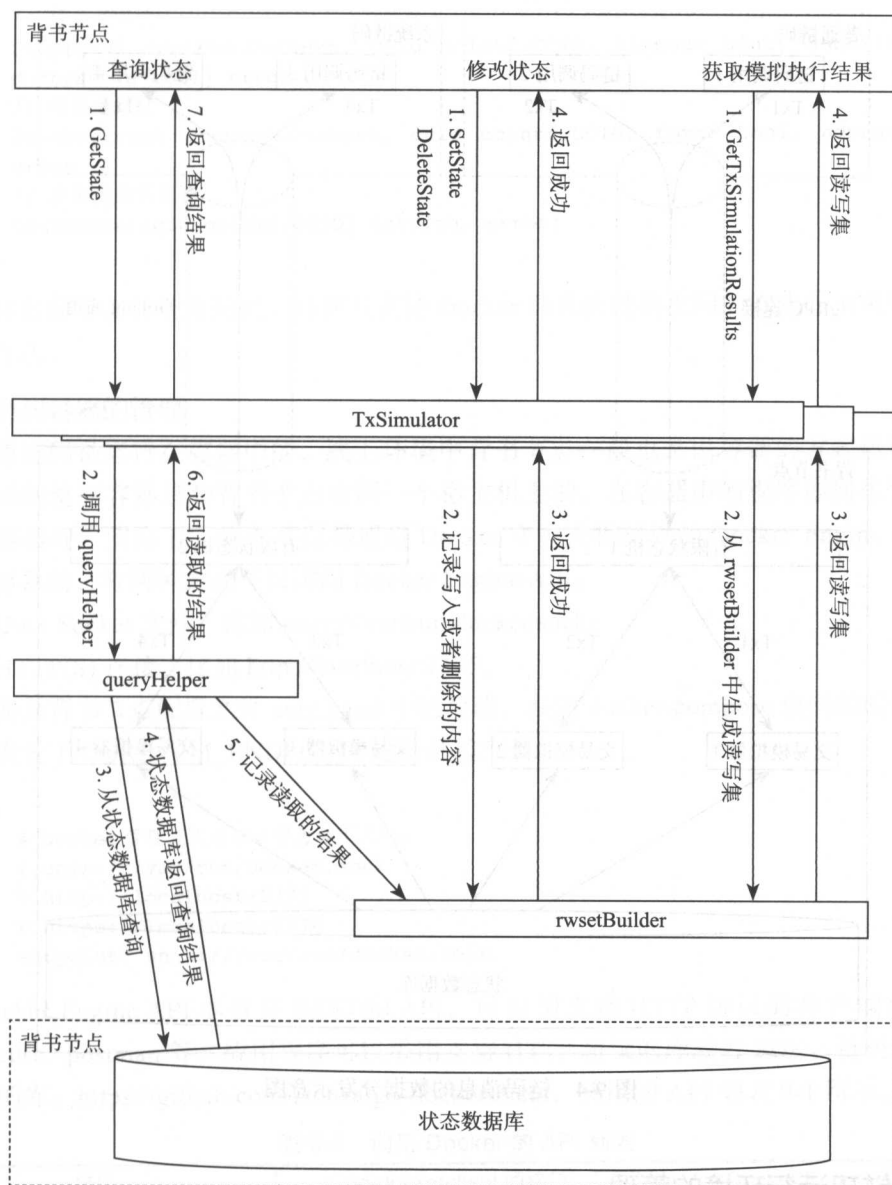


图 9-3 交易模拟器的运行示意图

图 9-4 是一个链码消息数据分发示意图，图中普通链码、系统链码分别和背书节点建立了不同的连接，普通链码建立的是 gRPC 连接，系统链码建立的是 Golang 的通道连接，同一个连接上又分别有 2 个链码调用在操作数据。背书节点侧的有限状态机记录了不同交易号调用时的上下文信息，根据交易号从交易上下文映射表中获取到交易所对应的上下文信息，进而利用交易模拟器生成模拟执行的结果。

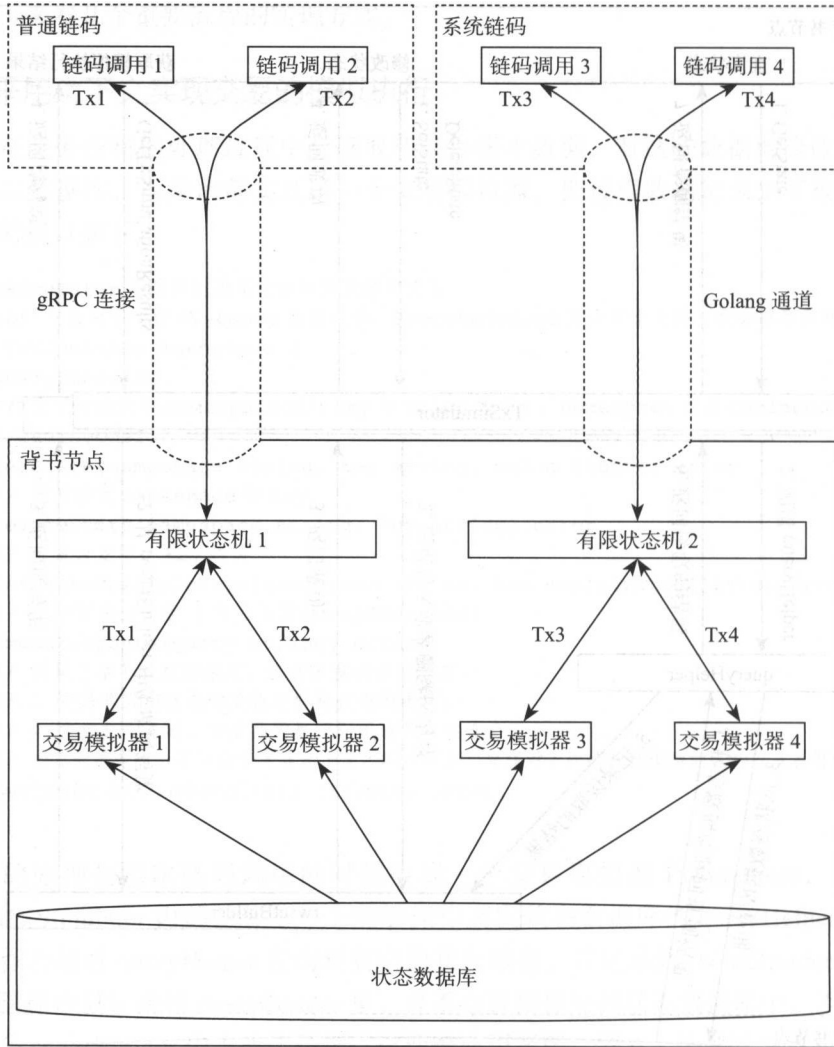


图 9-4 链码消息的数据分发示意图

### 9.2.6 链码运行环境的管理

链码运行环境管理需要实现的接口如下。

```
type VM interface {  
    // 部署虚拟机  
    Deploy(ctxt context.Context, ccid ccintf.CCID, args []string, env []string,  
    reader io.Reader) error  
    // 启动虚拟机  
    Start(ctxt context.Context, ccid ccintf.CCID, args []string, env []string,  
    builder BuildSpecFactory, preLaunchFunc PrelaunchFunc) error
```



```

// 停止虚拟机
Stop(ctxt context.Context, ccid ccintf.CCID, timeout uint, dontkill bool,
dontremove bool) error
// 销毁虚拟机
Destroy(ctxt context.Context, ccid ccintf.CCID, force bool, noprun bool)
error
// 获取虚拟机名称
GetVMName(ccID ccintf.CCID) (string, error)
}

```

接口名称定义的是虚拟机，目前只支持 Docker 和系统进程空间的方式，未来可能支持更多的方式。

### 1. 链码容器的管理

普通链码是运行在容器中的，线上环境中背书节点一般也是以容器的方式运行，背书节点启动的链码容器是和背书节点在同一个宿主机上的。在容器中的程序如何来管理宿主主机上的容器呢？实际上，背书节点是通过 Docker 守护进程提供的 Docker Engine API 创建和启动容器的，有两种方式可以访问 Docker Engine API：

- Unix Socket 文件，比如 `unix:///var/run/docker.sock`；
- HTTP(S) 连接，比如 `http://localhost:2375`。

下面是背书节点配置文件 `core.yaml` 中的片段，在用 `docker-compose` 启动的时候需要确认是否设置了 `CORE_VM_ENDPOINT` 这个环境变量。

```

vm:
  # Docker 管理方式可以设置为如下几种：
  # unix:///var/run/docker.sock
  # http://localhost:2375
  # https://localhost:2376
  endpoint: unix:///var/run/docker.sock

```

Docker Engine API 本身是 RESTful API，可以用支持 HTTP 协议的客户端访问，如 `wget`、`curl`、`postman` 等，应用程序可以采用支持 HTTP 协议的库或者 SDK。链码采用的是第三方的库：<https://github.com/fsouza/go-dockerclient>，调用的 API 如表 9-1 所示。

表 9-1 调用 Docker 的 API 列表

接口	go-dockerclient 的接口	Docker Engine API
Deploy	Client.BuildImage	POST /build
ListImages	Client.ListImages	GET /images/json
Start	Client.CreateContainer	POST /containers/create
Stop	Client.KillContainer	POST /containers/id/kill
Destroy	Client.RemoveImageExtended	DELETE /images/id
GetVMName	无	无

我们先来看下镜像的构建流程，如图 9-5 所示。

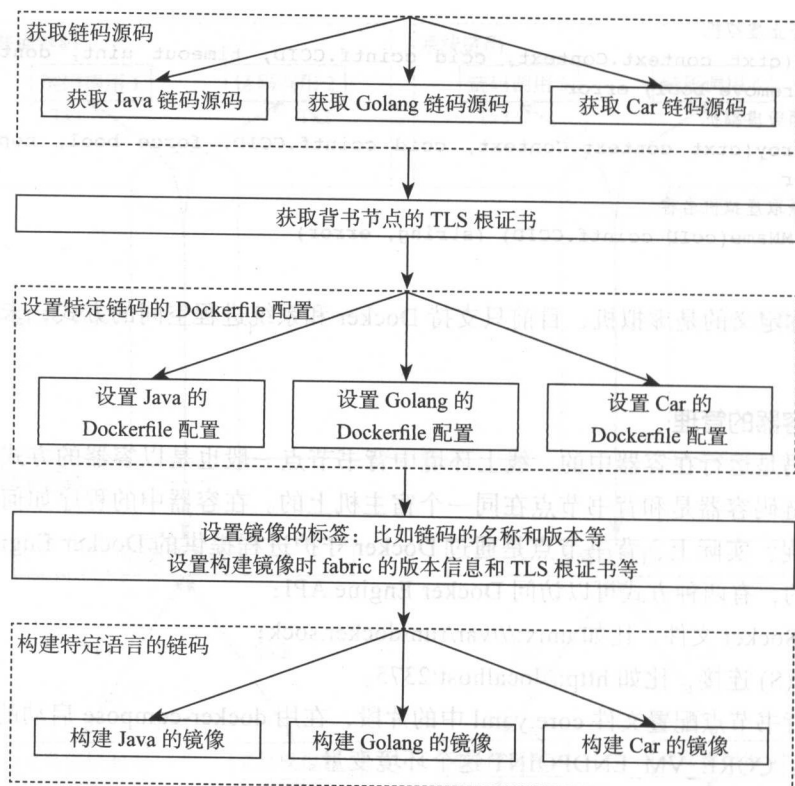


图 9-5 构建链码镜像示意图

特别说明一下，构建链码镜像的时候会写入背书节点 TLS 的根证书，链码节点根据这个根证书验证背书节点的 gRPC 连接是否是安全的。在镜像构建的时候如果已经存在同名的镜像，就不会重复构建。只要组织和链码名称、版本都相同，则这种情况就容易出现。尤其是本地搭建环境的时候，如果重新生成了 MSP 的证书，没有删除已生成的镜像就部署链码，就可能会导致链码容器里的 TLS 根证书和背书节点的 TLS 根证书不匹配，出现连接错误。链码在启动的时候会检查账本的链码信息是否和本地文件保存的链码文件信息一致，但并不检查链码容器镜像是否和账本的链码信息一致，其实是有隐藏的问题的。

构建链码镜像的过程是跟链码语言相关的，主要的功能是在源码的基础上添加连接背书节点的 SDK 和其他的一些依赖，再编译成可执行的二进制文件，编译环境是启动一个语言相关的临时容器，编译完成后拷贝生成的二进制文件到镜像文件里面。这个部分跟链码语言相关，比如 Golang 会增加环境变量 GOPATH 和编译参数等，细节的内容就不展开了，感兴趣的读者可以查看源码的 `core/chaincode/platform` 目录。

构建完的链码镜像会写入一些构建时的标签信息，使用 `docker inspect` 可查看到如下标签，如表 9-2 所示。

链码容器在编译和启动过程中会继承背书节点的一些环境变量，如表 9-3 所示。

表 9-2 获取 Docker 的标签信息列表

标签名称	说明
org.hyperledger.fabric.chaincode.id.name	链码的名称
org.hyperledger.fabric.chaincode.id.verison	链码的版本
org.hyperledger.fabric.chaincode.type	链码的语言
org.hyperledger.fabric.version	链码构建时 Faric 的版本
org.hyperledger.fabric.base.version	链码构建时基础镜像的版本

表 9-3 链码容器继承背书节点的环境变量列表

环境变量	继承时机	说明
CORE_PEER_TLS_ENABLED	启动	背书节点是否采用 TLS 传输
CORE_PEER_TLS_ROOTCERT_FILE	构建	背书节点的 TLS 根证书
CORE_PEER_TLS_SERVERHOSTOVERRIDE	启动	TLS 握手验证的服务端域名
CORE_CHAINCODE_LOGGING_LEVEL	启动	链码的日志级别
CORE_CHAINCODE_LOGGING_SHIM	启动	链码访问背书节点的 SDK 的日志级别
CORE_CHAINCODE_LOGGING_FORMAT	启动	链码的日志格式
CORE_CHAINCODE_BUILDLEVEL	编译	链码构建时 Fabric 的版本

启动链码的命令行参数 `--peer.address` 也是从背书节点的环境变量中获取的，这个命令行参数是链码和背书节点建立 gRPC 连接的服务端地址。有几个相关的环境变量：

- ❑ `CORE_PEER_CHAINCODELISTENADDRESS`;
- ❑ `CORE_PEER_ADDRESS`;
- ❑ `CORE_CHAINCODE_PEERADDRESS`。

这个几个环境变量的使用是有优先级的，如果设置了环境变量 `CORE_PEER_CHAINCODELISTENADDRESS`，则会以这个环境变量为准，否则以环境变量 `CORE_PEER_ADDRESS` 为准。最后一个环境变量 `CORE_CHAINCODE_PEERADDRESS` 是备用的，只有前面两个设置有异常的情况下才会启用。如果都没有设置或者设置有问题，就是默认值 `0.0.0.0:7051` 了。

2. 系统链码的管理

系统链码运行在 Peer 节点的进程空间中，实现方式比容器的方式简单许多。比如部署的过程实际就是在内存中注册名称和入口函数，标识系统链码在运行就可以了。启动的过程主要是建立好和背书节点的 Golang 通道，启动有限状态机对交互消息进行处理，后面的部分和容器的过程是一样的。系统链码的调用过程如图 9-6 所示。

系统链码的调用过程跟普通链码不同的地方在于不需要启动容器，只需要检查是否启动，根据链码运行时环境找到对应的 Golang 通道，通过通道发送调用链码的请求即可。系统链码调用的入口有两个，可以是节点进程内部发起调用，也可以是通过网络的方式调用。两种方式调用的流程并不一样，进程内部发起的调用执行完成以后不会调用 ESCC 进行背

书签名。只有部分系统链码可以外部调用，更详细的内容请看下面一个章节。

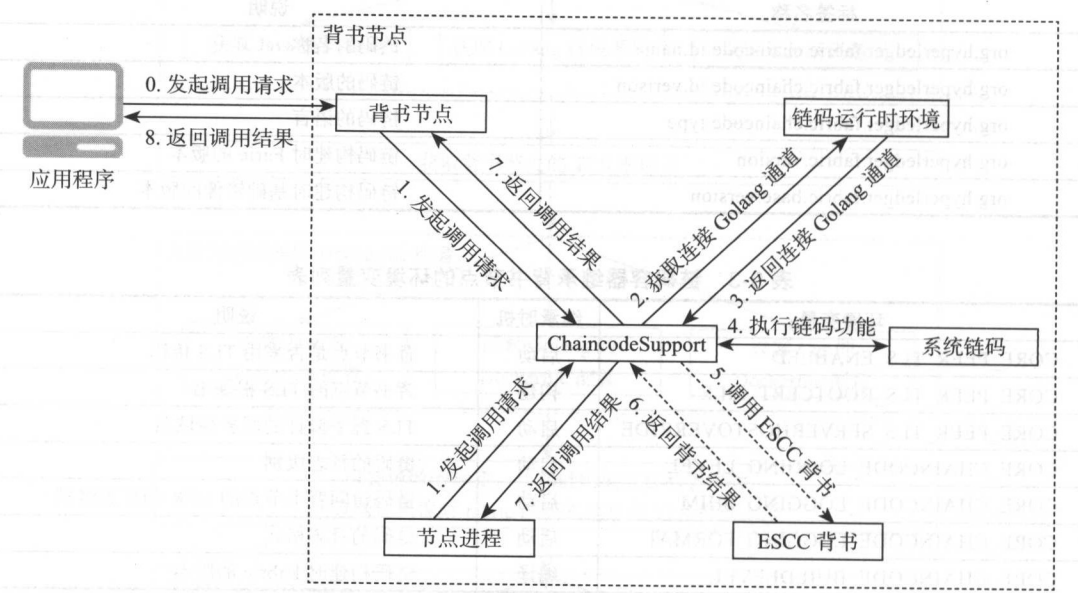


图 9-6 系统链码运行示意图

### 9.3 内置的系统链码

链码分为两种类型。

- ❑ 系统链码：系统内置的链码，用来完成一些系统功能等。
- ❑ 普通链码：实现应用业务逻辑的链码。

系统链码和普通链码的几个不同点，如表 9-4 所示。

表 9-4 系统链码和普通链码的不同点

对比项	系统链码	普通链码
链码源码	无 main 函数	有 main 函数
运行空间	背书节点进程	Docker
调用方式	网络 + 进程内部	网络
启动参数	内置	动态输入
通信方式	Golang 的通道机制	网络
数据存取	Golang 的通道 + 本地文件	网络
升级方式	和背书节点一起升级	单独升级
背书策略	无	有

下面是系统链码的属性定义。

```
type SystemChaincode struct {
```

```

//system chaincode 名称 (唯一)
Name string
//system chaincode 路径 (暂未使用)
Path string
//system chaincode 初始化参数
InitArgs [][]byte
// chaincode object 实体对象
Chaincode shim.Chaincode
// 用于追踪是否通过发送提案给 Peer 节点来完成对 system chaincode 调用
InvokableExternal bool
// 用于追踪是否可以通过 chaincode-to-chaincode 的调用方式调用 system chaincode
InvokableCC2CC bool
// 用于开启 / 禁用 system chaincode 的便捷开关, 无须删除 importsysccs.go 的条目
Enabled bool
}

```

以下是内置的系统链码及其属性列表, 如表 9-5 所示。

表 9-5 内置的系统链码及其属性列表

链码名称	路径	是否可外部调用	是否可被其他链码调用	是否可被替代
lscc	github.com/hyperledger/fabric/core/scc/lscc	是	是	否
cscc	github.com/hyperledger/fabric/core/scc/cscc	是	否	否
qscc	github.com/hyperledger/fabric/core/scc/qscc	是	是	否
escc	github.com/hyperledger/fabric/core/scc/escc	否	否	是
vscc	github.com/hyperledger/fabric/core/scc/vscc	否	否	是

下面就来详细说明一下每个系统链码的功能。

### 9.3.1 生命周期管理系统链码

生命周期管理系统链码 (Lifecycle System Chaincode, LSCC), 主要功能是管理部署在背书节点上的链码, 并不是全生命周期的管理。

#### 1. 链码的安装

链码安装接收的参数如下。

```

type ChaincodeDeploymentSpec struct {
    // 链码描述规范
    ChaincodeSpec *ChaincodeSpec
    // 控制链码可用事件, 预留字段
    EffectiveDate *google_protobuf1.Timestamp
    // 打包好的链码源代码
    CodePackage []byte
    // 运行环境: 系统进程或者 Docker
    ExecEnv ChaincodeDeploymentSpec_ExecutionEnvironment
}

```

其中, CodePackage 是已经打包好的链码源代码, 会在源码基础上添加所需要的库文件, 比如和背书节点通信的 [github.com/hyperledger/fabric/core/chaincode/shim](https://github.com/hyperledger/fabric/core/chaincode/shim)。链码描述规范 ChaincodeSpec 约定链码部署或者调用的一些参数, 定义如下。

```
type ChaincodeSpec struct {
    // 链码类型 :Golang、Node.js、Java、Car (Chaincode Archive) , 目前只支持 Golang
    Type ChaincodeSpec_Type
    // 链码的路径、名称和版本信息
    ChaincodeID *ChaincodeID
    // 链码的输入参数
    Input *ChaincodeInput
    // 超时时间, 预留字段
    Timeout int32
}

type ChaincodeID struct {
    // 链码路径
    Path string
    // 链码名称
    Name string `protobuf:"bytes,2,opt,name=name" json:"name,omitempty"`
    // 链码版本
    Version string `protobuf:"bytes,3,opt,name=version" json:"version,omitempty"`
}

type ChaincodeInput struct {
    // 链码输入参数数组
    Args [][]byte
}
```

在调用 LSCC 的时候, ChaincodeDeploymentSpec 是 ChaincodeInput 的第 2 个参数, 第 1 个参数是固定的“install”。就是说外层还有一层封装, 封装在 ChaincodeInvocationSpec 中。

```
type ChaincodeInvocationSpec struct {
    // 链码描述规范
    ChaincodeSpec *ChaincodeSpec
    // 可包含指定用户 ID 的生成算法, 通过此方法来生成 ID。
    // 若不指定 (或留空), 将使用 sha256base64 算法。算法主要包含两部分:
    // 1, hash 函数; 2, 将用户的 (字符串) 输入进行字节解码
    // 目前, 支持带 BASE64 的 SHA256 (例如: idGenerationAlg='sha256base64')
    IdGenerationAlg string
}
```

其中, ChaincodeInvocationSpec 中的 ChaincodeSpec 指定的 ChaincodeID.Name 是固定的“lsc”, 消息结构图如图 9-7 所示。

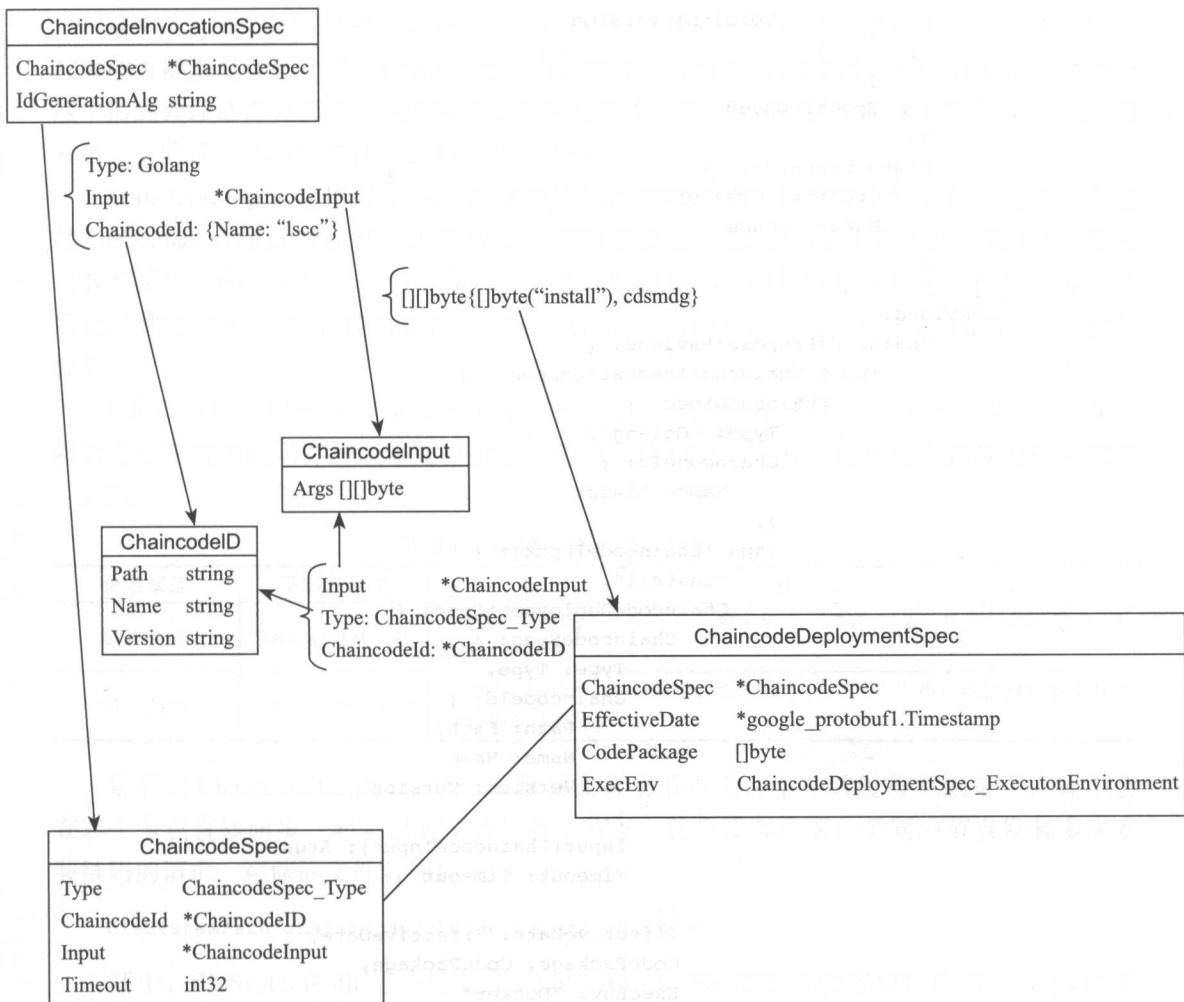


图 9-7 链码安装结构图

再加上必需的消息头，链码安装提交的 Proposal 逻辑结构如下。

```

SignedProposal: {
  ProposalBytes(Proposal): {
    Header: {
      ChannelHeader: {
        Type: "HeaderType_ENDORSER_TRANSACTION",
        TxId: TxId,
        Timestamp: Timestamp,
        Extension(ChaincodeHeaderExtension): {
          PayloadVisibility: PayloadVisibility,
          ChaincodeId: {
            Path: Path,
            Name: Name,

```



```

        Version: Version
    },
    Epoch: Epoch
},
SignatureHeader: {
    Creator: Creator,
    Nonce: Nonce
},
Payload: {
    ChaincodeProposalPayload: {
        Input(ChaincodeInvocationSpec): {
            ChaincodeSpec: {
                Type: "Golang",
                ChaincodeId: {
                    Name: "lsccl"
                },
            },
            Input(ChaincodeInput): {
                "install",
                ChaincodeDeploymentSpec: {
                    ChaincodeSpec: {
                        Type: Type,
                        ChaincodeId: {
                            Path: Path,
                            Name: Name,
                            Version: Version
                        },
                        Input(ChaincodeInput): Args,
                        Timeout: Timeout
                    },
                    EffectiveDate: EffectiveDate,
                    CodePackage: CodePackage,
                    ExecEnv: "Docker"
                },
            },
        },
    },
    TransientMap: TransientMap
},
Signature: Signature
}

```

其中，有"()"的地方，比如 ProposalBytes (Proposal) 中，ProposalBytes 代表字段名称，Proposal 代表实际的类型。有""的是请求固定的内容，其他的变量会根据实际的请求发生变化。TxId 是交易号，是在客户端构建 Proposal 的时候生成的，生成规则是：

```
TxID = HASH(Nonce + Creator)
```

其中, `Nonce` 是随机数, `Creator` 是创建请求的用户身份信息, 独立生成的交易号出现冲突的可能性是比较小的。`Creator` 还有别的用处, 背书节点接收到请求以后 `LSCC` 会根据这个信息验证是否有权限安装链码或者其他的操作。`Signature` 是 `Creator` 对整个 `Proposal` 的签名, 能验证消息的完整性, 起到防抵赖的作用。

`TransientMap` 是应用程序提交给背书节点处理但不记录到账本中的数据, 链码可以通过 `GetTransient` 接口获取到, 用来传输一些敏感信息。比如, 链码传递的参数 `Input` 里的内容都是加密的, 密钥通过 `TransientMap` 传递给链码, 链码可以解密调用参数的内容, 最后记录到账本里的只有加密的参数, `TransientMap` 的内容在 `ESCC` 签名背书的时候是会删掉的。

`LSCC` 直接处理的是 `ChaincodeDeploymentSpec` 结构, 首先会根据签名信息验证权限, 然后会检查链码的名称和版本是否合法。链码名称和版本的命名都是有内置规范的, 如表 9-6 所示。

表 9-6 链码名称和版本的命名规范

命名规范	正则表达式	说明
链码名称	"[A-Za-z0-9_-]+"	允许大写字母、小写字母、数字和 “_” “-” 组成的任意长度字符串 (不能是空字符串)
链码版本	"[A-Za-z0-9_-].]+"	允许大写字母、小写字母、数字和 “_” “-” “.” 组成的任意长度字符串 (不能是空字符串)

最后把 `ChaincodeDeploymentSpec` 存储在背书节点文件中安装过程就结束了, 并不会在多个节点直接同步。如果想要安装到多个节点, 就需要客户端向不同的节点发起多次安装链码的请求。存储的文件名称是:

```
filePath/chaincodes/chaincodeName:chaincodeVersion
```

其中, `filePath` 是环境变量 `CORE_PEER_FILESYSTEMPATH` 或者 `core.yaml` 文件配置的 `peer.filePath` 定义的, `chaincodes` 是固定的子目录。从上面的目录结构可以看出, 链码存储的时候并没有区分不同链的命名空间, 实际多链是共享链码的。在链码部署和调用的时候会根据背书节点本地存储的链码构建链码镜像, 启动链码容器。

## 2. 链码的部署

链码部署提交的 `Proposal` 逻辑结构与安装的逻辑结构基本相同, 区别在于 `ChaincodeInvocationSpec.ChaincodeSpec.Input` 内容不一样。

```
Input(ChaincodeInvocationSpec): {
  "deploy",
  ChaincodeDeploymentSpec: {
    ChaincodeSpec: {
      Type: Type,
      ChaincodeId: {
        Path: Path,
```

```

        Name: Name,
        Version: Version
    },
    Input(ChaincodeInput): Args,
    Timeout: Timeout
},
EffectiveDate: EffectiveDate,
CodePackage: CodePackage,
ExecEnv: "Docker"
},
chainID,
policy(SignaturePolicyEnvelope): {
    Version: Version,
    Rule(SignaturePolicy): {
        Type: SignedBy/NOutOf
    },
    Identities(common1.MSPPrincipal): [
        PrincipalClassification: PrincipalClassification,
        Principal: Principal
    ]
},
"escc",
"vsc"
}

```

其中，第1个参数是"deploy"，后面增加了几个参数，"escc"和"vsc"默认是内置的系统链码，也可以自行实现替换。SignaturePolicyEnvelope的详细介绍参考成员管理章节的内容。

部署的时候会检查实例化策略，默认的实例化策略是管理员权限。最后会记录ChaincodeData到链上。

```

type ChaincodeData struct {
    // 链码名称
    Name string
    // 链码版本
    Version string
    // 链码的 ESCC，默认是内置的 escc
    Escc string
    // 链码的 VSCC，默认是内置的 vsc
    Vsc string
    // 链码的背书策略
    Policy []byte
    // 链码源代码哈希和元数据哈希组成的 CDSData
    Data []byte
    // 链码编号，预留字段
    Id []byte
    // 链码实例化策略
    InstantiationPolicy
}

```

```

type CDSData struct {
    // 链码源代码哈希=hash(chaincode)
    CodeHash []byte
    // 链码元数据哈希=hash(chaincodeName+chaincodeVersion)
    MetadataHash []byte
}

```

实例化只能调用执行一次，实例化的时候检查链上有 ChaincodeData 就说明已经实例化过了。

背书节点在调用 LSCC 执行完部署的工作后，还会继续执行实例化的操作。这种情况只有链码是 LSCC，并且调用是部署（deploy）和升级（upgrade）的时候才执行。就是说，只有 LSCC 的 Proposal 里的 ChaincodeInvocationSpec 才会有嵌套 ChaincodeDeploymentSpec 的结构。

链码的安装是和具体的链没有关系的，并不会记录数据到账本里，同一个链码是可以多个链共享的，实例化的时候才会记录到不同链的账本数据里，不同链的数据是独立隔离的。

### 3. 链码的升级

链码的升级和部署比较类似，也需要先安装链码，再执行链码的升级。链码升级需要保证链码的名称和升级之前的一样，否则会被当成另外一个链码。由于链码的版本可以是符合命名规范的字符串，并不是严格意义的递增关系，是以提交的先后顺序为准的，链码升级的时候会检查升级的版本是否和最新版本的版本号一致。与链码部署不同的地方在于，链码升级的时候会检查链码最新版本的实例化策略，再执行链码的更新，更新的过程中也会执行实例化的操作。

链码升级并不会删除老版本的链码，多个版本是可以并存的，需要手工维护应用程序和链码之间的对应关系，避免调用出现错误。链码升级的操作过程参考前面介绍的内容。

### 4. 链码信息的查询

可以查询如下几个接口的信息，如表 9-7 所示。

表 9-7 链码信息查询的接口列表

查询接口	查询内容	说明
getid	查询链码的名称	返回链码的名称，会校验一下链上是否存在 ChaincodeData 信息
getdepspec	查询链码的部署信息规范 Chaincode DeploymentSpec	返回 chaincodeInstallPath 目录下存储的 Chaincode DeploymentSpec 信息，会校验链上存储的 Chaincode Data 信息是否和本地存储的一致
getccdata	查询链码的基本信息 ChaincodeData	返回链上存储的 ChaincodeData 信息
getchaincodes	查询背书节点实例化的所有链码	从 LSCC 链上查询有 ChaincodeData 信息的链码
getinstalledchaincodes	查询背书节点安装的所有链码	遍历 chaincodeInstallPath 目录下所有的链码，同一个链码的不同版本也会返回

### 9.3.2 配置管理系统链码

配置管理系统链码（CSCC）的全称是 Configuration System Chaincode，主要功能是管理记账节点上的配置信息。

#### 1. 记账节点加入链

记账节点加入链的 Proposal 请求的 ChaincodeInput 不再是多层的嵌套了，结构如下。

```
SignedProposal: {
  ProposalBytes(Proposal): {
    Header: {
      ChannelHeader: {
        Type: "HeaderType_CONFIG",
        TxId: TxId,
        Timestamp: Timestamp,
        Extension(ChaincodeHeaderExtension): {
          PayloadVisibility: PayloadVisibility,
          ChaincodeId: {
            Path: Path,
            Name: Name,
            Version: Version
          }
        },
        Epoch: Epoch
      },
      SignatureHeader: {
        Creator: Creator,
        Nonce: Nonce
      }
    },
    Payload: {
      ChaincodeProposalPayload: {
        Input(ChaincodeInvocationSpec): {
          ChaincodeSpec: {
            Type: "Golang",
            ChaincodeId: {
              Name: "csc"
            },
            Input(ChaincodeInput): {
              "JoinChain",
              genesisBlock
            }
          }
        },
        TransientMap: TransientMap
      }
    },
    Signature: Signature
  }
}
```

其中，genesisBlock 是创建通道时生成的创世区块。CSCC 的 Peer 节点加入通道的过程如图 9-8 所示。

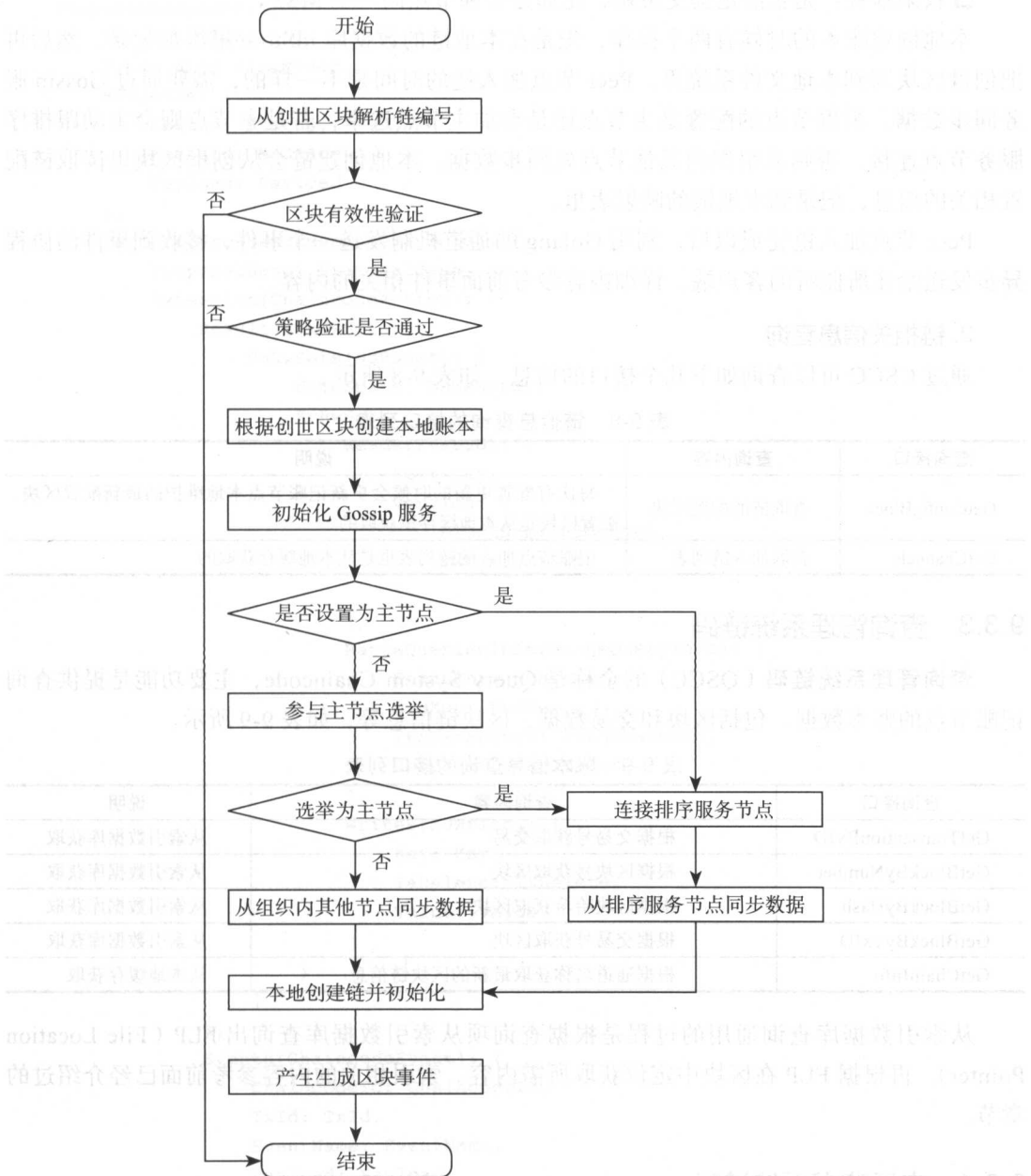


图 9-8 Peer 节点加入通道流程图

其中，区块的有效性验证主要包括如下几个方面。

□ 类型检查：查看区块类型是否是 HeaderType\_CONFIG。

□ 配置检查：查看配置区块里是否包含 Application 信息。

□ 权限检查：是否满足提交策略，比如是否和节点同一个 MSP。

本地创建账本的时候有两个操作，先是在本地链的数据库 idStore 里添加记录，然后再把创世区块写到本地文件系统里。Peer 节点加入链的时间是不一样的，需要通过 Gossip 服务同步数据，根据节点的配置是主节点还是参加主节点选举，若是主节点则会主动跟排序服务节点连接，否则从组织内其他节点处同步数据。本地创建链会从创世区块里读取链配置相关的信息，记录到本地链的映射表里。

Peer 节点加入链完成以后，利用 Golang 的通道机制发送一个事件，接收到事件的协程序异步发送给注册监听的客户端。详细内容参考前面事件相关的内容。

### 2. 链相关信息查询

通过 CSCC 可以查询如下几个接口的信息，如表 9-8 所示。

表 9-8 链信息查询的接口列表

查询接口	查询内容	说明
GetConfigBlock	查询链的配置区块	每次有配置更新的时候会更新记账节点本地维护的最新配置区块，配置区块是从本地缓存里获取的
GetChannels	获取加入链列表	记账节点加入的链列表也是从本地缓存获取的

### 9.3.3 查询管理系统链码

查询管理系统链码（QSCC）的全称是 Query System Chaincode，主要功能是提供查询记账节点的账本数据，包括区块和交易数据、区块链信息等，如表 9-9 所示。

表 9-9 账本信息查询的接口列表

查询接口	查询内容	说明
GetTransactionByID	根据交易号获取交易	从索引数据库获取
GetBlockByNumber	根据区块号获取区块	从索引数据库获取
GetBlockByHash	根据区块哈希获取区块	从索引数据库获取
GetBlockByTxID	根据交易号获取区块	从索引数据库获取
GetChainInfo	根据通道名称获取最新的区块链信息	从本地缓存获取

从索引数据库查询通用的过程是根据查询项从索引数据库查询出 FLP（File Location Pointer），再根据 FLP 在区块中定位获取所需内容。索引相关的内容参考前面已经介绍过的章节。

### 9.3.4 交易背书系统链码

交易背书系统链码（ESCC）的全称是 Endorsement System Chaincode，主要功能是对交易进行结果的结构转换和签名背书。



ESCC 是背书节点模拟执行完链码后，对执行的结果 Response 进行转换，去掉里面的 Transient 信息，再添加签名背书，最后封装成 ProposalResponse，结构如下。

```
ProposalResponse: {
    Version: Version,
    Timestamp: Timestamp,
    Response: {
        Status: Status,
        Message: Message,
        Payload: Payload
    },
    Payload(ProposalResponsePayload): {
        ProposalHash: ProposalHash,
        Extension(ChaincodeAction): {
            Results(TxRwSet): {
                NsRwSets(NsRwSet): [
                    Namespace: Namespace,
                    KvRwSet: {
                        Reads(KVRead): [
                            Key: Key,
                            Version: {
                                BlockNum: BlockNum,
                                TxNum: TxNum
                            }
                        ],
                        RangeQueriesInfo(RangeQueryInfo): [
                            StartKey: StartKey,
                            EndKey: EndKey,
                            ItrExhausted: ItrExhausted,
                            ReadsInfo: ReadsInfo
                        ],
                        Writes(KVWrite): [
                            Key: Key,
                            IsDelete: IsDelete,
                            Value: Value
                        ]
                    }
                ]
            },
            Events(ChaincodeEvent): {
                ChaincodeId: ChaincodeId,
                TxId: TxId,
                EventName: EventName,
                Payload: Payload
            }
        },
        Response: {
            Status: Status,
```

```

        Message: Message,
        Payload: Payload
    },
    ChaincodeId: ChaincodeId
}
},
Endorsement: {
    Endorser: Endorser,
    Signature: Signature
}
}

```

其中, Endorsement 就是背书的内容, 包含背书节点的签名证书 Endorser 和背书节点对 ProposalResponsePayload+Endorser 的签名, 即:

```
Signature=Endorser.Sign(ProposalResponsePayload+Endorser)
```

背书代表背书节点用自己的身份对模拟执行结果进行担保, 通过证书和签名就可以验证背书是否是有效的。

### 9.3.5 交易验证系统链码

交易验证系统链码 (VSCC) 的全称是 Validation System Chaincode, 主要功能是记账前对区块和交易进行验证。ESCC 是背书节点独立对模拟执行结果的背书, VSCC 是对多个背书验证是否符合背书策略。VSCC 验证交易背书的过程参考第 3 章的交易背书策略部分。

## 9.4 链码的相互调用

在第 5 章中, 我们已经介绍过, 不同链的账本数据和状态数据等都是物理隔离或者逻辑隔离的。对于同一个链不同链码的状态数据, 会按链码名称生成不同前缀的键, 对状态数据进行逻辑隔离。不同的链码实现不同的业务逻辑, 是可以相互调用的。调用的方法通过 shim.InvokeChaincode:

```
InvokeChaincode(chaincodeName string, args [][]byte, channel string) pb.Response
```

其中, chaincodeName 是被调用链码的名称, 链码名称可以指定版本, 比如 mycc:1.0 这样的形式, args 是被调用链码的参数, channel 是被调用链码的通道名称, 默认调用的是同一个链的链码。链码名称会规范化处理, 生成的链码名称是:

```
chaincodeName:chaincodeVersion/channelName
```

其中的链码版本 chaincodeVersion 和通道名称 channelName 都是可选的, 对应的分隔符也是可选的, 所以链码名称可能有如下几种组合, 如表 9-10 所示。

表 9-10 链码相互调用的名称列表

链码名称格式	链码名称示例	链码名称说明
chaincodeName	mycc	调用相同链的链码 mycc, 调用的链码版本是最新版本
chaincodeName/channelName	mycc/mychannel	调用链名称为 mychannel 的链码 mycc, 调用的链码版本是最新版本
chaincodeName:chaincodeVersion	mycc:1.0	调用相同链的链码 mycc, 调用的链码版本是 1.0
chaincodeName:chaincodeVersion/channelName	mycc:1.0/mychannel	调用链名称为 mychannel 的链码 mycc, 调用的链码版本是 1.0

从上面链码的规范化名称, 我们可以看到, 链码的相互调用分为以下两种情况:

- ☐ 同一个链的链码相互调用;
- ☐ 不同链的链码相互调用。

同一个链或者不同链的调用流程是基本一样的, 调用 `shim.InvokeChaincode` 后会构造一个类型为 `ChaincodeMessage_INVOKE_CHAINCODE` 的 `ChaincodeMessage`, 消息是通过链码和背书节点之间的 gRPC 连接直接提交给背书节点的有限状态机处理的, 不是通过背书节点接收 Proposal 的背书流程。背书节点对链码相互调用的处理过程如图 9-9 所示。

不同链的链码相互调用不同的地方在于会生成一个新的交易模拟器 `TxSimulator`, 实现对被调用链数据的访问, 最终生成交易的读写集只会包含调用链的数据, 并不会修改被调用链的状态数据。如果是调用相同链的链码, 会复用相同的交易模拟器 `TxSimulator`, 链码执行的结果会修改最终的状态数据。

## 9.5 背书节点和链码的有限状态机

链码本身是不会存储任何数据的, 业务逻辑处理过程中是通过建立好的 gRPC 连接实现和背书节点的交互, 交互过程是通过有限状态机 (Finite State Machine) 来实现的。有限状态机有下面几个特点:

- ☐ 状态是有限的, 能够遍历完所有的状态;
- ☐ 有一个初始状态和终止状态以及若干中间状态;
- ☐ 任意时刻只会处于其中的一个状态;
- ☐ 处于某个状态下能处理的事件是有限的;
- ☐ 状态转移之间的转移条件是确定的。

背书节点端和链码端都通过有限状态机定义了各自生命周期内所处的所有状态, 以及如何在各种状态下响应各种事件和转移到其他状态。具体实现采用第三方的库 <http://github.com/looplab/fsm>, 我们就用 `fsm` 来代表这个库。`fsm` 的状态直接用字符串来表示, 定义了状态转移映射表。

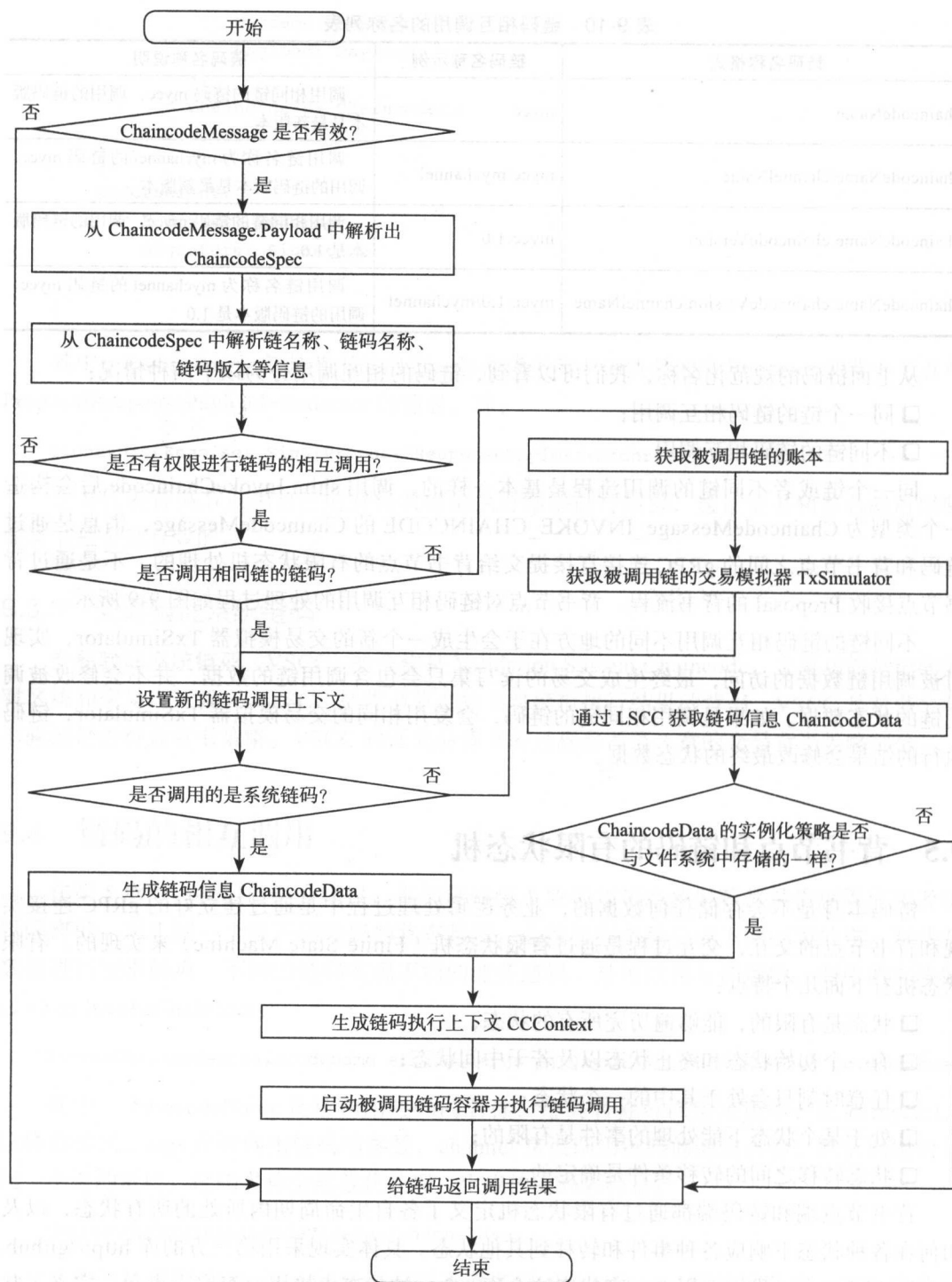


图 9-9 背书节点对链码相互调用的处理过程图

```

type EventDesc struct {
    // 事件名称
    Name string
    // 状态转移的源状态列表
    Src []string
    // 状态转移的目的状态
    Dst string
}

```

其中，状态转移的源状态列表 Src 是一个数组，是把状态转移合并了，多个状态都可以接收相同的事件转移到相同的目的状态。

回调函数映射表 Callbacks 定义了事件处理和状态转移的执行函数。

```

type Callbacks map[string]Callback

```

```

type Callback func(*Event)

```

```

type Event struct {
    // 对当前 FSM 的引用
    FSM *FSM
    // 事件名称
    Event string
    // 交易前的状态
    Src string
    // 交易后的状态
    Dst string
    // 回调函数中返回的错误 (可选)
    Err error
    // 回调函数传入的参数 (可选)
    Args []interface{}
    // 数字标识位，当交易取消时设定
    canceled bool
    // 数字标识位，当异步交易时设定
    async bool
}

```

回调函数映射表 Callbacks 定义了两种类型的函数。

□ 事件处理的执行函数：处理某个事件前后的操作，定义规则是 before\_EVENT 和 after\_EVENT，其中 EVENT 是某个具体的事件名称。

□ 状态变化的执行函数：进入某个状态和离开某个状态的操作，定义规则是 enter\_STATE 和 leave\_STATE，其中 STATE 是某个具体的状态名称。

有限状态机通用的事件处理流程如图 9-10 所示。

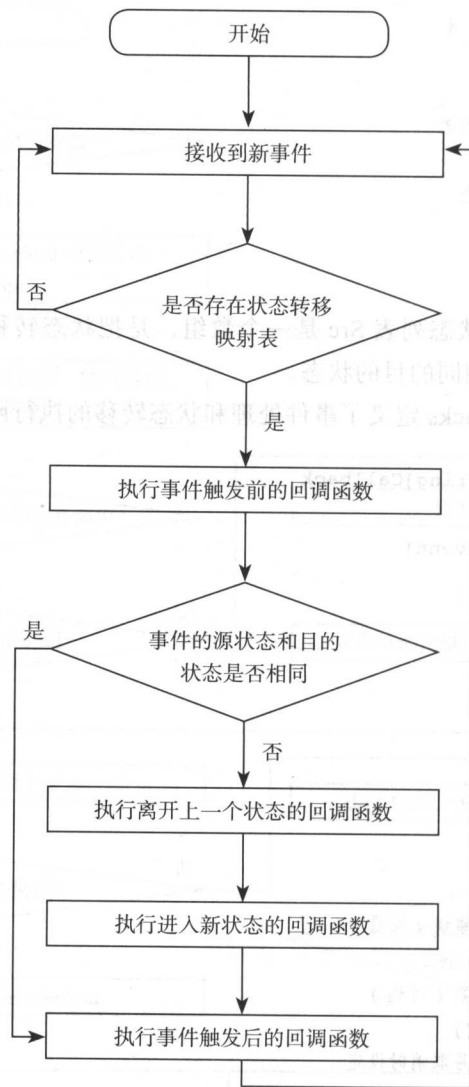


图 9-10 有限状态机处理流程图

9.5.1 背书节点和链码之间的事件

背书节点和链码之间的事件列表，如表 9-11 所示。

表 9-11 背书节点和链码之间的事件列表

事件名称	发起方	接收方	说明
UNDEFINED	背书节点	链码	预留的未定义消息，用来做异常的错误处理
REGISTER	链码	背书节点	链码启动的时候发送给背书节点的注册消息，用来建立连接

(续)

事件名称	发起方	接收方	说明
REGISTERED	背书节点	链码	链码在背书节点注册完成以后的确认消息
INIT	背书节点	链码	背书节点发送给链码的初始化消息, 会调用链码的 Init 接口
READY	背书节点	链码	背书节点的初始化工作完成, 等待接收链码消息请求并处理
TRANSACTION	背书节点	背书节点 链码	背书节点发送给链码的部署或者调用请求
COMPLETED	链码	背书节点 链码	链码初始化完成
ERROR	背书节点 链码	背书节点 链码	错误处理消息
GET_STATE	链码	背书节点	链码发起的 GetState 请求
PUT_STATE	链码	背书节点	链码发起的 PutState 请求
DEL_STATE	链码	背书节点	链码发起的 DelState 请求
INVOKE_CHAINCODE	链码	背书节点	链码发起的 InvokeChaincode 请求
RESPONSE	背书节点 链码	背书节点 链码	背书节点或者链码处理请求返回的结果
GET_STATE_BY_RANGE	链码	背书节点	链码发起的 GetStateByRange 请求
GET_QUERY_RESULT	链码	背书节点	链码发起的 GetQueryResult 请求
QUERY_STATE_NEXT	链码	背书节点	链码发起的 Next 迭代查询请求
QUERY_STATE_CLOSE	链码	背书节点	链码发起的 Close 迭代查询结束请求
KEEPALIVE	背书节点	链码	背书节点发起的心跳消息
GET_HISTORY_FOR_KEY	链码	背书节点	链码发起的 GetHistoryForKey 请求

## 9.5.2 背书节点的有限状态机

背书节点的状态列表, 如表 9-12 所示。

表 9-12 背书节点的状态列表

状态名称	说明
created	初始状态
established	注册以后的状态
ready	运行状态, 能够正常接收消息请求并处理
end	终止状态

如图 9-11 所示, 背书节点的有限状态机进入 ready 的运行状态后, 就可以接收链码的请求了, 主要是和状态数据操作相关的接口, 比如状态数据的更新 PUT\_STATE、DEL\_STATE, 状态数据的 GET\_STATE、GET\_STATE\_BY\_RANGE 等。背书节点接收到链码的请求后, 通过交易模拟器实现状态数据的读写, 返回给链码进行业务逻辑的处理, 实现链码的无状态部署和运行。



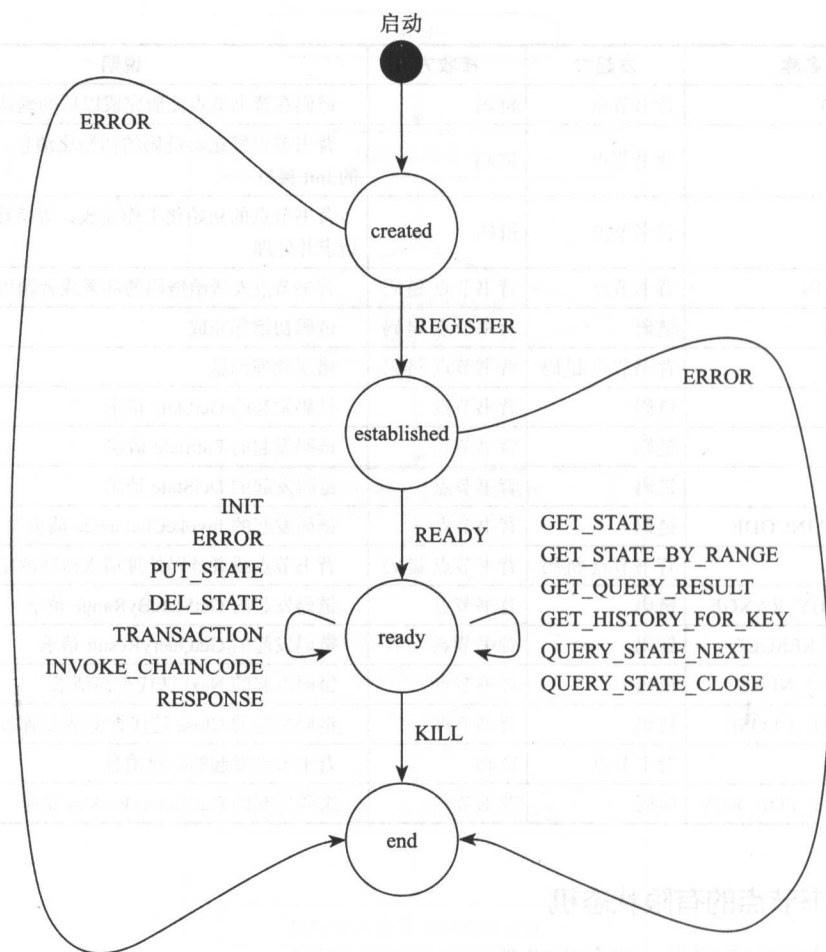


图 9-11 背书节点的有限状态机

### 9.5.3 链码的有限状态机

链码的状态列表，如表 9-13 所示。

表 9-13 链码的状态列表

状态名称	说明
created	初始状态
established	注册以后的状态
init	初始化状态
ready	运行状态，能够正常接收消息请求并处理
end	终止状态

链码是主动给背书节点发起的请求，只需要接收背书节点返回的数据就可以了，所以

链码侧的有限状态机比较简单，如图 9-12 所示。

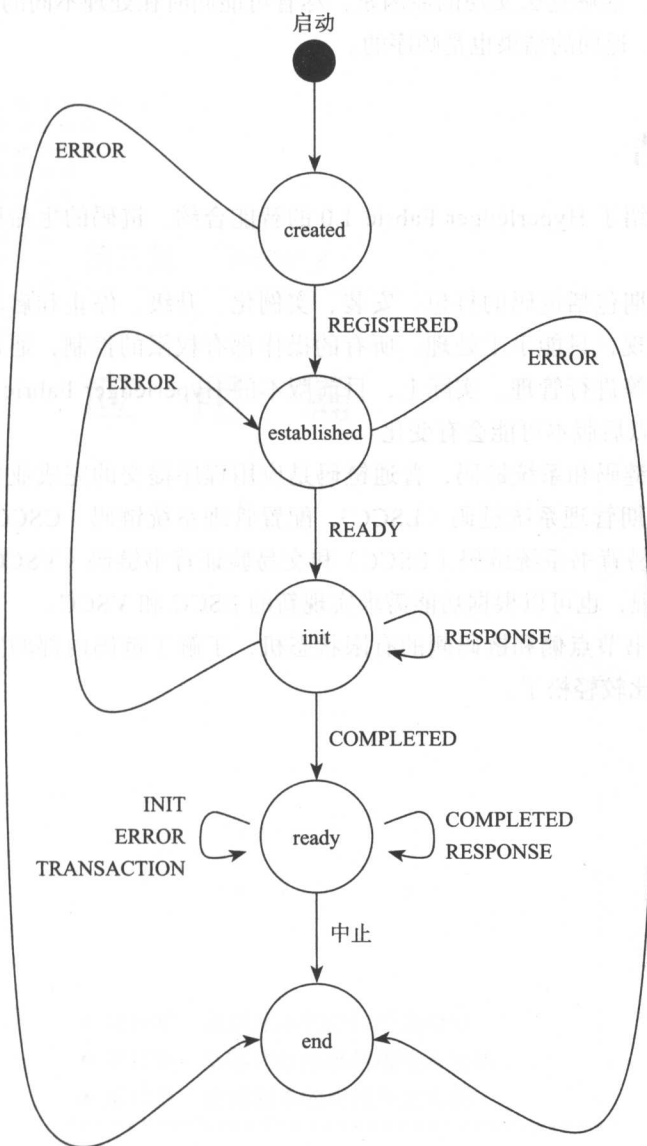


图 9-12 链码的有限状态机

同一个链码也可能同时和背书节点进行交互，链码侧也会存在背书节点侧消息的数据分发问题。链码维护了交易的 Golang 通道 `responseChannel` 映射表。

```
responseChannel map[string]chan pb.ChaincodeMessage
```

其中，键是交易号 `txid`，值是不同交易号的 Golang 通道 `pb.ChaincodeMessage`。链码有

限状态机接收到 ChaincodeMessage\_RESPONSE 的消息后, 通过给每个交易建立的 Golang 通道返回给调用接口。能够这么实现的原因是, 尽管可能同时在处理不同的交易, 但是同一个交易是顺序执行的, 返回的结果也是顺序的。

## 9.6 本章小结

本章详细地介绍了 Hyperledger Fabric 1.0 的智能合约, 链码的生命周期管理及其实现原理。

链码的生命周期包括链码的打包、安装、实例化、升级、停止和启动, 其中链码的停止和启动还没有实现, 只能手工处理。所有的操作都有权限的控制, 通过通道策略、实例化策略和背书策略等进行管理。实际上, 目前版本的 Hyperledger Fabric 1.0 实现的权限控制都是粗粒度的, 以后版本可能会有变化。

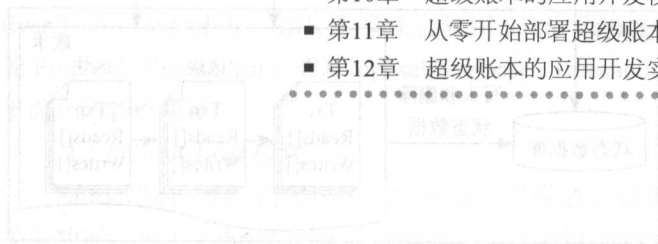
链码分为普通链码和系统链码, 普通链码是应用程序提交的完成业务功能的链码。系统链码包括生命周期管理系统链码 (LSCC)、配置管理系统链码 (CSCC)、查询管理系统链码 (QSCC)、交易背书系统链码 (ESCC) 和交易验证背书链码 (VSCC), 其中 ESCC 和 VSCC 有默认的实现, 也可以根据功能需求实现新的 ESCC 和 VSCC。

最后介绍了背书节点侧和链码侧的有限状态机, 了解了链码内部的实现原理, 理解下一章的链码接口就比较轻松了。

## 第三篇 *Part 3*

# 应用篇

- 第10章 超级账本的应用开发模型
- 第11章 从零开始部署超级账本网络
- 第12章 超级账本的应用开发实例



# 超级账本的应用开发模型

前面章节介绍的都是 Hyperledger Fabric 1.0 内部的机制和原理，不做任何源码级别的改动就可以部署起来，提供基本的区块链底层平台服务。本章会从应用的角度出发，介绍如何开发基于 Fabric 网络的区块链应用。

## 10.1 应用开发模型

我们从程序开发角度来看看各个模块的交互，首先应用程序接收用户的请求，然后可能调用智能合约，也可能直接访问区块链。智能合约在执行的过程中可能对区块链进行操作，并产生事件。Hyperledger Fabric 1.0 的应用开发模型如图 10-1 所示。

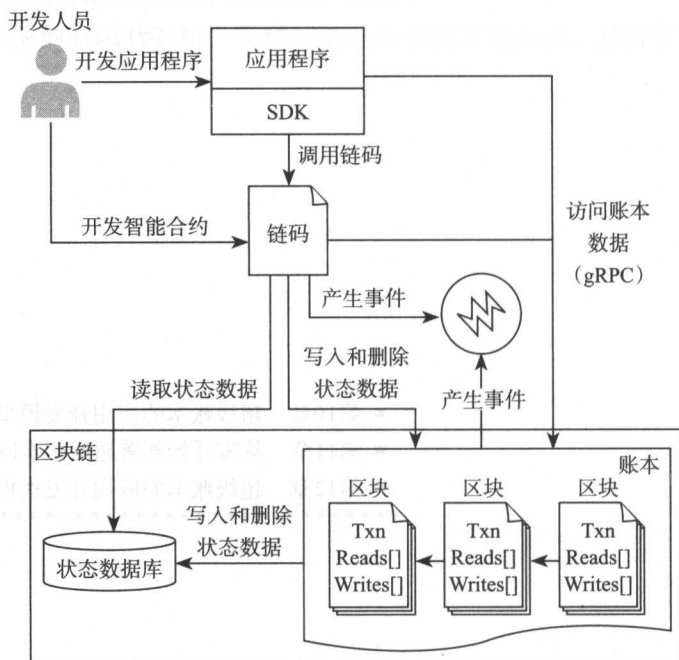


图 10-1 Hyperledger Fabric 1.0 的应用开发模型

## 10.2 应用程序开发的 SDK

本节先介绍应用开发 SDK 的基本功能。

### 10.2.1 概述

HFC (Hyperledger Fabric Client) 是提供给应用程序开发的 SDK, 提供了 gRPC 连接的 API。API 包含了交易处理、安全的成员管理服务、区块链查询和事件处理等。Hyperledger Fabric 1.0 取消了 0.6 版本的 RESTful 接口, 只能选择封装了 gRPC 接口的 SDK。采用 gRPC 的原因主要有以下四点。

- 底层的接口调用很多都是异步返回结果的, 采用 gRPC 能够很好地双向传输数据。
- gRPC 结合 Protocol Buffers 能减少传输数据量, 提升网络传输性能。
- 支持的语言较多, 如 Go、C#、Java、JavaScript、Python、C++ 等。
- 和内部模块采用相同的通信接口, 减少端口开放, 也会减少安全的风险。

整个 Fabric 网络中, 除了可选的 fabric-ca 采用的是 RESTful 接口之外, 其他所有组件之间的通信都采用 gRPC 接口。

### 10.2.2 SDK 规范

SDK 定义了两种模块的接口: 一个是访问 fabric-ca 的接口, 一个是访问 Fabric 的接口。其中 fabric-ca 模块是可选的, 可以选用其他成熟的第三方 CA 系统。官方提供了如下几种语言的 SDK 实现。

- Golang: <https://github.com/hyperledger/fabric-sdk-go>。
- Node.js: <https://github.com/hyperledger/fabric-sdk-node>。
- Python: <https://github.com/hyperledger/fabric-sdk-py>。
- Java: <https://github.com/hyperledger/fabric-sdk-java>。

下面以 Golang 为主介绍一下 SDK 的设计和实现。

#### 1. 访问 Fabric 模块介绍

我们先来看一下 Fabric 模块的 UML 图, 如图 10-2 所示。

主要的模块包括 FabricClient、Config、Channel、Peer、Orderer、User、KeyValueStore、EventHub、Logger 等, SDK 本身还会复用 fabric 源码提供的功能。还有一些其他的模块 (比如 Proposal、SignedProposal、ProposalResponse、Transaction、CryptoSuite 等) 在这里没有展示。下面介绍各模块接口。

##### (1) FabricClient 模块

FabricClient 是应用程序的入口模块, 提供通道管理、链码管理、数据存储、密码学相关的功能。每个 FabricClient 实例对应一个区块链的网络, 包括记账节点、排序节点等。如果应用程序需要访问多个网络, 可以建立多个 FabricClient 的实例, 不同的实例对应不同的网络。FabricClient 模块的接口说明如表 10-1 所示。

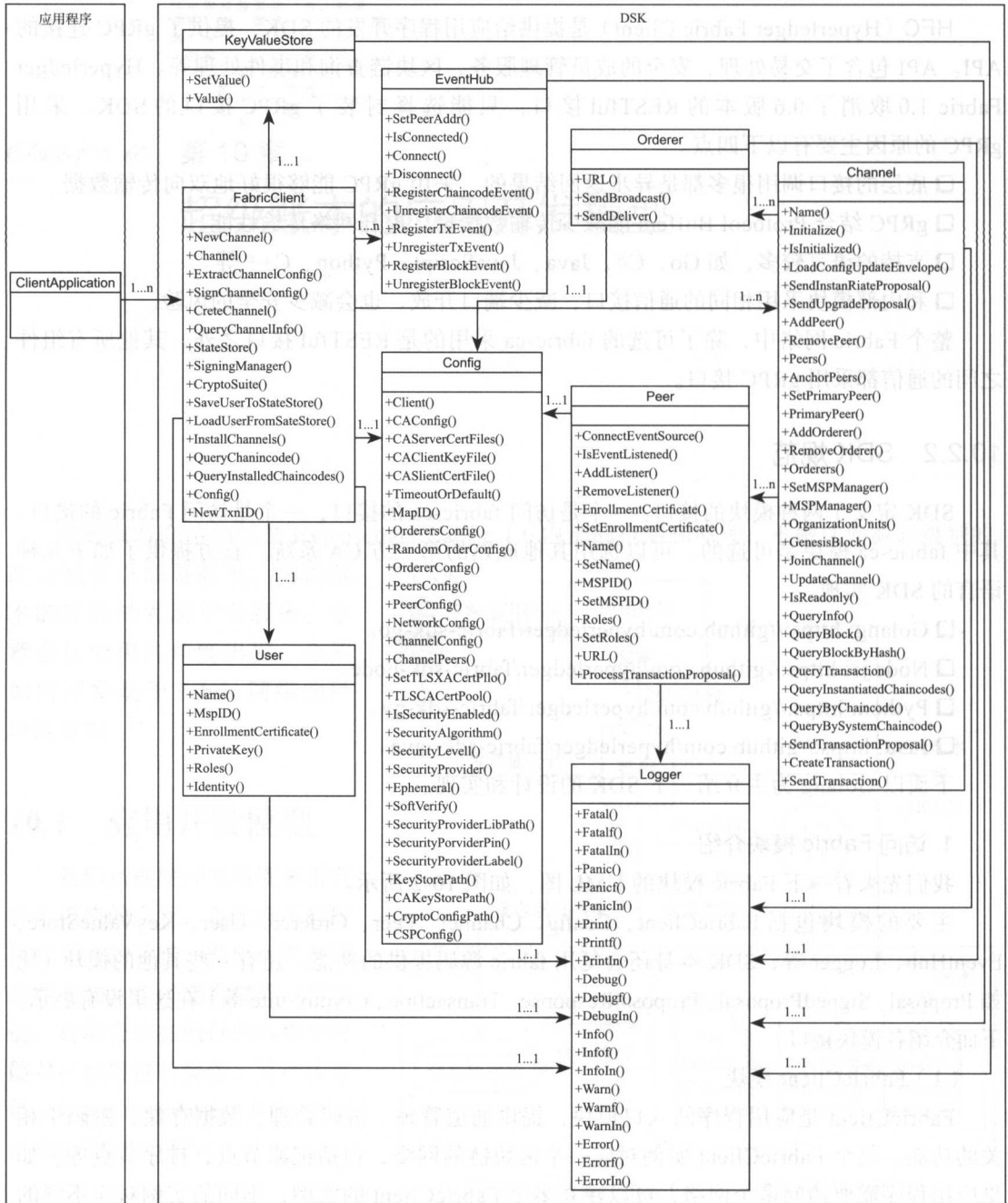


图 10-2 SDK 中访问 fabric 模块的 UML 图



表 10-1 FabricClient 模块的接口列表

接口名称	输入参数	输出参数	说明
NewChannel	(name string)	(Channel, error)	创建通道
Channel	(name string)	(Channel)	查询指定名称的通道
ExtractChannelConfig	(configEnvelope []byte)	([]byte, error)	从 ConfigEnvelope 里解析出 ConfigUpdate
SignChannelConfig	(config []byte)	(*common.ConfigSignature, error)	用 FabricClient 关联的用户身份对 ExtractChannelConfig 解析出来的 config 进行签名
CreateChannel	(request fab.CreateChannelRequest)	(apitxn.TransactionID, error)	创建通道, 创建通道的参数包括通道名称、排序服务实例、通道配置等信息, 返回包含随机数的交易号
QueryChannelInfo	(name string, peers []fab.Peer)	(fab.Channel, error)	从指定节点查询通道, 目前没有实现
StateStore	()	(fab.KeyValueStore)	返回状态存储的实例
SigningManager	()	(fab.SigningManager)	返回签名 manager 实例
CryptoSuite	()	(bccsp.BCCSP)	返回 BCCSP 实例
SaveUserToStateStore	(user fab.User, skipPersistence bool)	(error)	保存用户实例到状态存储里
LoadUserFromStateStore	(name string)	(fab.User, error)	从状态存储里获取指定名称的用户实例
InstallChaincode	(chaincodeName string, chaincodePath string, chaincodeVersion string, chaincodePackage []byte, targets []fab.Peer)	([]*apitxn.TransactionProposalResponse, string, error)	安装指定链码名称、路径、版本的链码到指定的节点中
QueryChannels	(peer fab.Peer)	(*pb.ChannelQueryResponse, error)	查询指定节点加入的所有通道
QueryInstalledChaincodes	(peer fab.Peer)	(*pb.ChannelQueryResponse, error)	查询指定节点安装的所有链码
UserContext	()	(user fab.User)	返回当前 FabricClient 的用户实例
SetUserContext	(user fab.User)	()	设置当前 FabricClient 的用户实例
Config	()	(config.Config)	设置当前 FabricClient 的配置实例
NewTxnID	()	(apitxn.TransactionID, error)	本地生成包含随机数的交易号

## (2) Config 模块

初始化 FabricClient 的时候需要离线获取配置信息, 包括可信的根证书、排序服务节点证书和 IP 地址、记账节点证书和 IP 地址等, 配置模块 Config 读取后传递给 FabricClient。

配置信息是动态传递的，SDK 不会持久化存储，应用程序负责维护这些配置信息。Config 模块的接口说明如表 10-2 所示。

表 10-2 Config 模块的接口列表

接口名称	输入参数	输出参数	说明
Client	()	(*ClientConfig, error)	返回 FabricClient 的配置实例
CAConfig	(org string)	(*CAConfig, error)	返回指定组织的 CA 配置实例
CAServerCertFiles	(org string)	([]string, error)	返回指定组织的 CA 证书文件路径
CAClientKeyFile	(org string)	(string, error)	返回指定组织 CA 的 TLS 私钥文件路径
CAClientCertFile	(org string)	(string, error)	返回指定组织 CA 的 TLS 证书文件路径
TimeoutOrDefault	(TimeoutType)	(time.Duration)	返回指定连接类型的超时事件
MspID	(org string)	(string, error)	返回指定组织的 MSP 名称
OrderersConfig	()	([]OrdererConfig, error)	返回通道的配置实例列表
RandomOrdererConfig	()	([]OrdererConfig, error)	返回随机个数通道的配置实例列表
OrdererConfig	(name string)	([]OrdererConfig, error)	返回指定通道名称的配置实例列表
PeersConfig	(org string)	([]PeerConfig, error)	返回指定组织名称的节点配置实例列表
PeerConfig	(org string, name string)	(*PeerConfig, error)	返回指定组织名称和节点名称的节点配置实例
NetworkConfig	()	(*NetworkConfig, error)	返回 Client 的网络配置实例
ChannelConfig	(name string)	(*ChannelConfig, error)	返回指定通道名称的配置实例
ChannelPeers	(name string)	([]ChannelPeer, error)	返回指定通道名称的节点配置实例
SetTLSCACertPool	(*x509.CertPool)	()	设置 Client 的全局 TLS 证书池
TLSCACertPool	(tlsCertificate string)	(*x509.CertPool, error)	返回指定证书文件的 TLS 证书池
IsSecurityEnabled	()	(bool)	查询是否开启安全模式
SecurityAlgorithm	()	(string)	查询安全模式的算法
SecurityLevel	()	(int)	查询安全模式的级别
SecurityProvider	()	(string)	查询安全模式的提供商
Ephemeral	()	(bool)	查询安全模式生成的是否是临时密钥
SoftVerify	()	(bool)	查询安全模式是否是软验证
SecurityProviderLibPath	()	(string)	返回安全供应商的库路径 (只对 PKCS11 供应商有效)
SecurityProviderPin	()	(string)	返回安全供应商设备的密码 (只对 PKCS11 供应商有效)
SecurityProviderLabel	()	(string)	返回安全供应商设备的标签 (只对 PKCS11 供应商有效)
KeyStorePath	()	(string)	返回密钥存储路径 + “keystore” 子目录 (对 PKCS11 供应商无效)
CAKeyStorePath	()	(string)	返回密钥存储路径 (对 PKCS11 供应商无效)

(续)

接口名称	输入参数	输出参数	说明
CryptoConfigPath	()	(string)	返回用户证书密钥路径
CSPConfig	()	(*bccspFactory.FactoryOpts)	返回 CSP 的配置

### (3) Channel 模块

通道是排序服务创建的隔离不同链上交易的实例，加入到不同通道的节点接收到的是不同的交易。通道在配置了排序服务节点和 Peer 节点后需要初始化，初始化的时候给排序服务节点发送获取配置区块的请求。Channel 模块的接口说明如表 10-3 所示。

表 10-3 Channel 模块的接口列表

接口名称	输入参数	输出参数	说明
Name	()	(string)	返回通道的名称
Initialize	(data []byte)	(error)	根据排序服务节点获取到配置初始化通道
IsInitialized	()	(bool)	查询通道是否已经初始化
LoadConfigUpdateEnvelope	(data []byte)	(error)	从配置中加载通道
SendInstantiateProposal	(chaincodeName string, args byte, chaincodePath string, chaincodeVersion string, chaincodePolicy *common.SignaturePolicyEnvelope, targets []txn.ProposalProcessor)	([]*txn.TransactionProposalResponse, txn.TransactionID, error)	发送链码实例化的请求给多个背书节点
SendUpgradeProposal	(chaincodeName string, args byte, chaincodePath string, chaincodeVersion string, chaincodePolicy *common.SignaturePolicyEnvelope, targets []txn.ProposalProcessor)	([]*txn.TransactionProposalResponse, txn.TransactionID, error)	发送链码升级的请求给多个背书节点
AddPeer	(peer Peer)	(error)	通道中增加一个 Peer 节点
RemovePeer	(peer Peer)	()	通道中删除一个 Peer 节点
Peers	()	([]Peer)	返回通道的 Peer 节点列表
AnchorPeers	()	([]OrgAnchorPeer)	返回通道组织的锚节点列表
SetPrimaryPeer	(peer Peer)	(error)	设置查询的主节点
PrimaryPeer	()	(Peer)	返回查询的主节点
AddOrderer	(orderer Orderer)	(error)	通道中增加一个排序服务节点
RemoveOrderer	(orderer Orderer)	()	通道中删除一个排序服务节点
Orderers	()	([]Orderer)	返回通道中的排序服务节点列表

(续)

接口名称	输入参数	输出参数	说明
SetMSPManager	(mspManager msp.MSP Manager)	()	设置通道的 MSPManager
MSPManager	()	(msp.MSPManager)	返回通道的 MSPManager
OrganizationUnits	()	([]string, error)	返回通道的组织列表
GenesisBlock	(request *GenesisBlockRequest)	(*common.Block, error)	从排序服务节点获取创世区块
JoinChannel	(request *JoinChannelRequest)	(error)	指定的 Peer 节点根据排序服务节点获取的创世区块加入通道
UpdateChannel	()	(bool)	更新通道的配置, 目前暂时没有实现
IsReadonly	()	(bool)	返回通道是否已经终止, 目前暂时没有实现
QueryInfo	()	(*common.BlockchainInfo, error)	返回通道的区块链信息
QueryBlock	(blockNumber int)	(*common.Block, error)	返回指定区块号的区块
QueryBlockByHash	(blockHash []byte)	(*common.Block, error)	返回指定区块哈希的区块
QueryTransaction	(transactionID string)	(*pb.ProcessedTransaction, error)	返回指定交易号的交易
QueryInstantiated Chaincodes	()	(*pb.ChaincodeQuery Response, error)	返回通道已经实例化的链码
QueryByChaincode	(txn.ChaincodeInvoke Request)	(byte, error)	链码的查询
SendTransactionProposal	(ChaincodeInvokeRequest)	([]*TransactionProposalResponse, TransactionID, error)	发送请求给背书节点
CreateTransaction	(resps []*Transaction ProposalResponse)	(*Transaction, error)	根据背书结果创建交易
SendTransaction	(tx *Transaction)	(*TransactionResponse, error)	发送交易给排序服务节点

#### (4) Peer 模块

Peer 节点是 HFC 模块发送背书请求、交易查询的节点。Peer 实例包含了节点名称、地址、角色、注册证书 (ECert) 等信息。Peer 模块的接口说明如表 10-4 所示。

表 10-4 Peer 模块的接口列表

接口名称	输入参数	输出参数	说明
ConnectEventSource	()	()	连接事件源, 目前暂未实现
IsEventListened	(event string, channel Channel)	(bool, error)	查询是否监听了指定通道上的指定事件, 目前暂未实现
AddListener	(eventType string, eventTypeData interface{}, eventCallback interface{})	(string, error)	给指定事件类型增加回调函数, 目前暂未实现

(续)

接口名称	输入参数	输出参数	说明
RemoveListener	(eventListenerRef)	(bool, error)	停止监听指定的事件
EnrollmentCertificate	()	(*pem.Block)	返回 Peer 节点的注册证书
SetEnrollmentCertificate	(pem *pem.Block)	()	设置 Peer 节点的注册证书
Name	()	(string)	返回 Peer 节点的名称
SetName	(name string)	()	设置 Peer 节点的名称
SetMSPID	()	(string)	设置 Peer 节点的 MSPID
Roles	()	([]string)	返回 Peer 节点的角色列表
SetRoles	(roles []string)	()	设置 Peer 节点的角色列表
URL	()	(string)	返回 Peer 节点的地址
ProcessTransactionProposal	(proposal TransactionProposal)	(TransactionProposal Result, error)	发送 Proposal 给 Peer 节点

### (5) Orderer 模块

Orderer 节点是 HFC 模块发送交易进行排序的节点。Orderer 实例包含了排序服务节点地址信息, 定义了发送原子广播请求和获取区块的接口。Orderer 模块的接口说明如表 10-5 所示。

表 10-5 Orderer 模块的接口列表

接口名称	输入参数	输出参数	说明
NewOrderer	(url string, certificate string, serverHostOverride string, config apiconfig.Config)	(*Orderer, error)	创建 Orderer 实例
URL	()	(string)	返回排序服务节点的地址
SendBroadcast	(envelope *SignedEnvelope)	(*common.Status, error)	发送交易请求给排序服务节点
SendDeliver	(envelope *SignedEnvelope)	(chan *common.Block, chan error)	返回指定请求生成的区块

### (6) User 模块

User 代表了已经生成注册证书和签名密钥的实体, 注册证书必须是区块链网络信任的 CA 颁发的证书, 只有生成了注册证书的实体才能进行部署链码、提交交易和查询交易等操作。注册证书可以从第三方 CA 获取, 也可以通过 fabric-ca 模块获取。

特别说明一下, 用户身份 (User Identity) 和节点身份 (Peer Identity) 是有区别的。在 SDK 里面, 用户身份能访问私钥信息, 是可以进行签名的。而节点身份不能访问私钥, 只能验证签名。

User 模块的接口说明如表 10-6 所示。

表 10-6 User 模块的接口列表

接口名称	输入参数	输出参数	说明
Name	()	(string)	返回用户的名称
MspID	()	(string)	返回用户的 MSPID

(续)

接口名称	输入参数	输出参数	说明
EnrollmentCertificate	()	([]byte)	返回用户的注册证书
PrivateKey	()	(bccsp.Key)	返回用户的私钥
Roles	()	([]string)	返回用户的角色列表
Identity	()	([]byte, error)	返回用户的身份信息 (包含证书和 MSPID)

### (7) KeyValueStore 模块

KeyValueStore 提供给应用程序保存敏感信息的功能, 比如用户私钥、证书信息等。KeyValueStore 模块的接口说明如表 10-7 所示。

表 10-7 KeyValueStore 模块的接口列表

接口名称	输入参数	输出参数	说明
SetValue	(key string, value []byte)	(error)	设置 key 的值为 value
Value	(key string)	([]byte, error)	查询 key 的值

### (8) EventHub 模块

EventHub 封装了与 Peer 节点交互的事件流, 接收 Peer 的各种异步通知事件。EventHub 模块的接口说明如表 10-8 所示。

表 10-8 EventHub 模块的接口列表

接口名称	输入参数	输出参数	说明
NewEventHub	(client fab.FabricClient)	(*EventHub, error)	创建 NewEventHub 实例
SetPeerAddr	(peerURL string, certificate string, serverHostOverride string)	([]byte, error)	设置事件源
IsConnected	()	(bool)	监测是否连接到事件源
Connect	()	(bool)	连接事件源
Disconnect	()	(bool)	断开事件源
RegisterChaincodeEvent	(ccid string, eventname string, callback func(*ChaincodeEvent))	([*ChainCodeCBE]	注册链码事件处理回调函数
UnregisterChaincodeEvent	(cbe *ChainCodeCBE)	()	注销链码事件处理回调函数
RegisterTxEvent	(txnID apitxn.TransactionID, callback func(string, pb.TxValidation Code, error))	()	注册交易事件处理回调函数
UnregisterTxEvent	(txnID apitxn.TransactionID)	()	注销交易事件处理回调函数
RegisterBlockEvent	(callback func(*common.Block))	()	注册区块事件处理回调函数
UnregisterBlockEvent	(callback func(*common.Block))	()	注销区块事件处理回调函数

### (9) Logger 模块

Logger 是日志模块, 提供了不同的日志接口, 基本都是日常开发过程中用到的通用日志模块, 这里就不详细展开了。



## 2. 访问 fabric-ca 模块介绍

我们再来看一下 fabric-ca 模块的 UML 图，如图 10-3 所示。

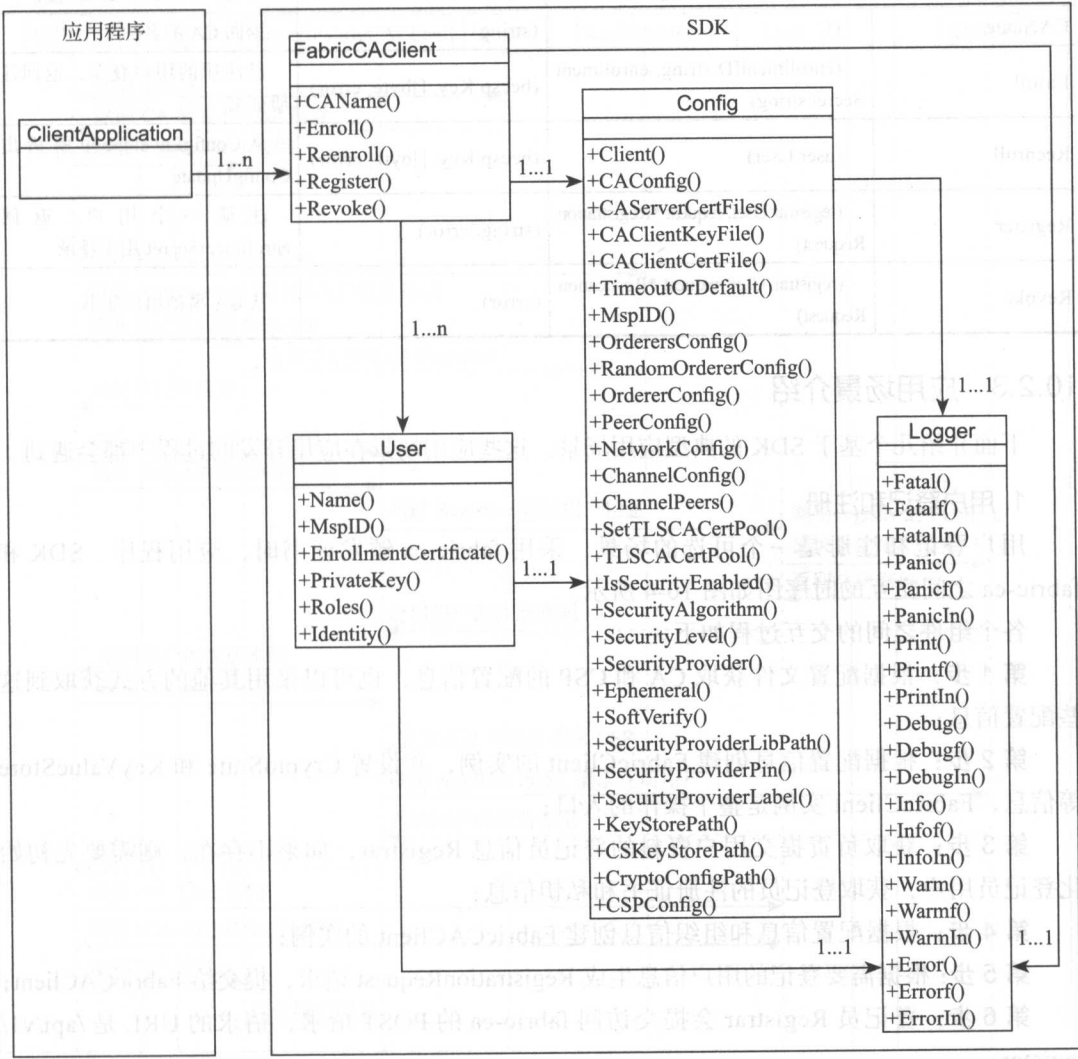


图 10-3 SDK 中访问 fabric-ca 模块的 UML 图

这里主要介绍 FabricCAClient 模块，其他的模块在前面已经介绍过。

FabricCAClient 是应用程序的入口模块，提供通道管理、链码管理、数据存储、密码学相关的功能。每个 FabricCAClient 实例对应一个区块链的网络，包括记账节点、排序节点等。如果应用程序需要访问多个网络，可以建立多个 FabricCAClient 的实例，不同的实例对应不同的网络。FabricCAClient 模块的接口说明如表 10-9 所示。



表 10-9 FabricCAClient 模块的接口列表

接口名称	输入参数	输出参数	说明
NewFabricCAClient	(config config.Config, org string)	(*FabricCA, error)	创建 FabricCAClient 实例
CAName	()	(string)	返回 CA 的名称
Enroll	(enrollmentID string, enrollment Secret string)	(bccsp.Key, []byte, error)	已注册的用户登录, 返回注册证书
Reenroll	(user User)	(bccsp.Key, []byte, error)	从 ConfigEnvelope 里解析出 ConfigUpdate
Register	(registrar User, request *Registration Request)	(string, error)	注册一个用户, 返回 enrollmentSecret 用于登录
Revoke	(registrar User, request *Revocation Request)	(error)	从 CA 吊销用户证书

10.2.3 应用场景介绍

下面介绍几个基于 SDK 的典型应用场景, 这些应用场景在应用开发的过程中都会遇到。

1. 用户登记和注册

用户登记和注册是一个可选的场景, 采用 fabric-ca 颁发证书时, 应用程序、SDK 和 fabric-ca 之间交互的时序图如图 10-4 所示。

各个组件之间的交互过程如下。

- 第 1 步: 根据配置文件获取 CA 和 CSP 的配置信息, 也可以采用其他方式获取到这些配置信息;
- 第 2 步: 根据配置信息创建 FabricClient 的实例, 并设置 CryptoSuite 和 KeyValueStore 等信息, FabricClient 实例是整个操作的入口;
- 第 3 步: 获取负责提交用户资料的登记员信息 Registrar, 如果不存在, 则需要先初始化登记员用户, 获取登记员的注册证书和私钥信息;
- 第 4 步: 根据配置信息和组织信息创建 FabricCAClient 的实例;
- 第 5 步: 根据需要登记的用户信息生成 RegistrationRequest 请求, 提交给 FabricCAClient;
- 第 6 步: 登记员 Registrar 会提交访问 fabric-ca 的 POST 请求, 请求的 URL 是 /api/v1/register;
- 第 7 步: fabric-ca 验证请求生成用户注册的密码 Secret, 最终返回给应用程序, 完成用户信息登记的步骤;
- 第 8 步: 应用程序利用申请的用户信息和返回的注册密码, 调用 FabricCAClient 的 Enroll 接口;
- 第 9 步: FabricCAClient 生成私钥和证书签名请求 CSR (Certificate Signing Request), 调用 fabric-ca 提供的 enroll 接口生成注册证书;
- 第 10 步: 返回生成的注册证书和私钥给应用程序;

第 11 步：可选的保存用户信息到 KeyValueStore 里。

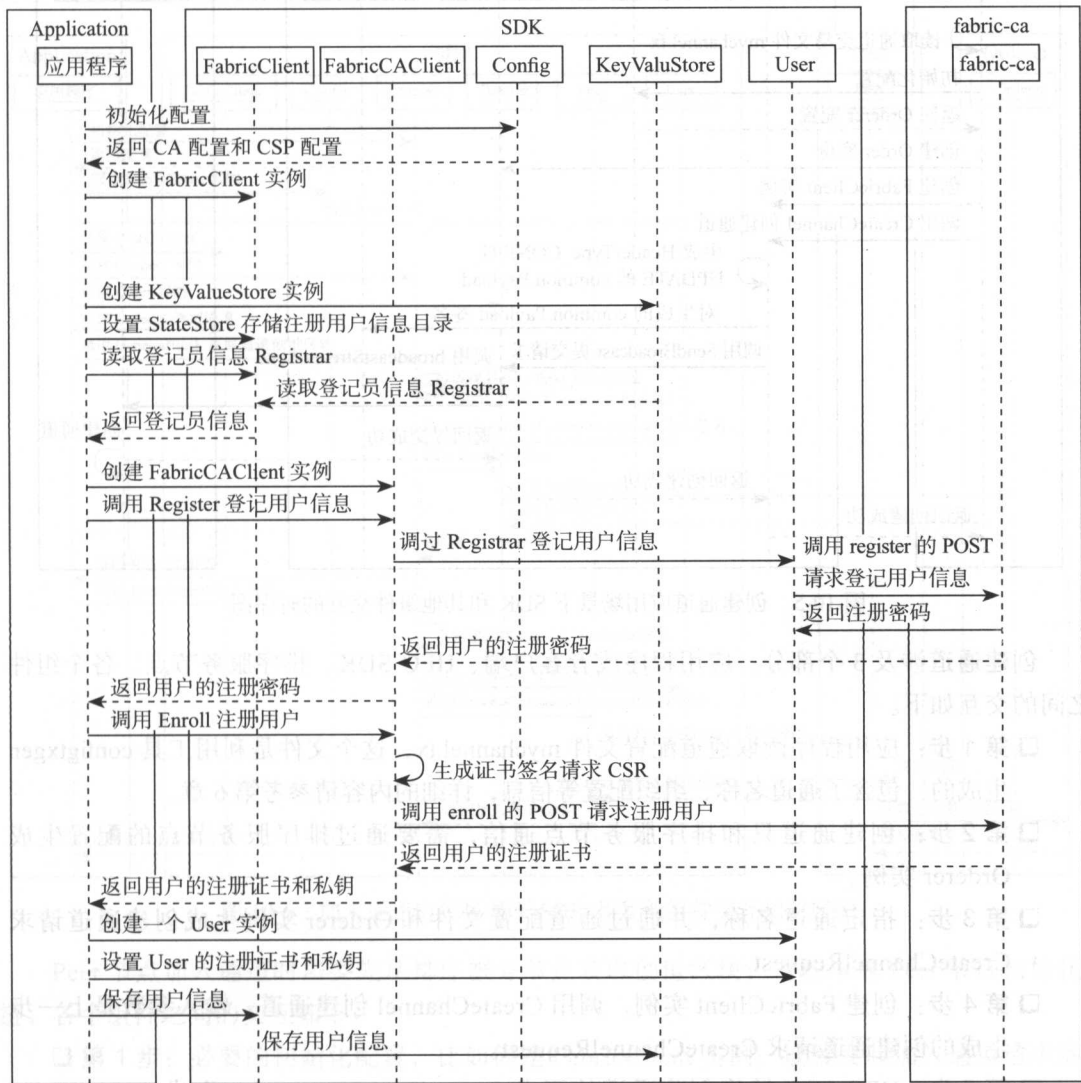


图 10-4 用户登记和注册应用场景下 SDK 和其他组件交互的时序图

fabric-ca 还提供了重新注册生成注册证书和吊销证书的功能，目前 fabric 并不能完整地支持 CRL 和 OCSP 的功能。

## 2. 在排序服务上创建通道

创建通道需要先利用工具 configtxgen 生成通道配置文件 mychannel.tx。创建通道应用场景下 SDK 和其他组件交互的时序图如图 10-5 所示。

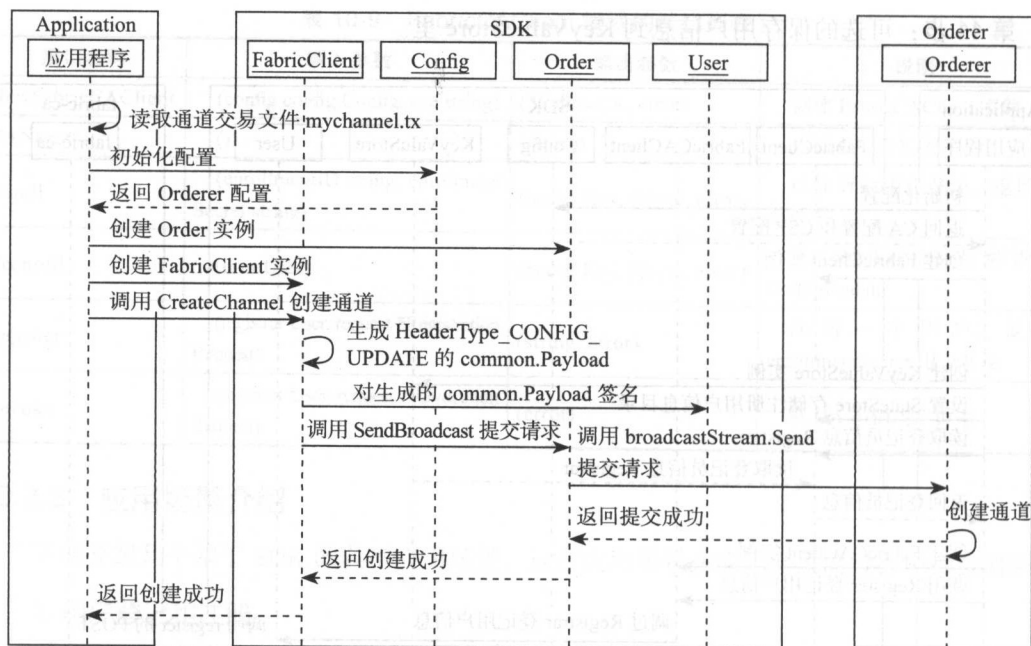


图 10-5 创建通道应用场景下 SDK 和其他组件交互的时序图

创建通道涉及 3 个部分，应用程序或者客户端、HFC SDK、排序服务节点。各个组件之间的交互如下。

- ❑ 第 1 步：应用程序读取通道配置文件 mychannel.tx，这个文件是利用工具 configtxgen 生成的，包含了通道名称、组织配置等信息，详细的内容请参考第 6 章。
- ❑ 第 2 步：创建通道只和排序服务节点通信，需要通过排序服务节点的配置生成 Orderer 实例。
- ❑ 第 3 步：指定通道名称，并通过通道配置文件和 Orderer 实例生成创建通道请求 CreateChannelRequest。
- ❑ 第 4 步：创建 FabricClient 实例，调用 CreateChannel 创建通道，输入参数是上一步生成的创建通道请求 CreateChannelRequest。
- ❑ 第 5 步：HFC SDK 转换创建通道的请求 CreateChannelRequest，生成 HeaderType\_CONFIG\_UPDATE 类型的交易 common.Payload。
- ❑ 第 6 步：HFC SDK 对 common.Payload 进行签名，签名者需要有通道创建的管理员权限。
- ❑ 第 7 步：通过 Orderer 实例发送 SendBroadcast 请求，提交请求给排序服务节点。
- ❑ 第 8 步：排序服务节点会检查提交的请求，校验是否有权限创建新的通道，创建通道以后排序服务节点就可以接收新通道的请求了。详细的内容请参考第 6 章。

### 3. Peer 节点加入通道

创建通道完成以后，排序服务节点上就有了新通道的基本信息，可以对新通道的交易

进行排序打包生成区块了。下一步需要把 Peer 加入到新通道中，应用程序或者客户端才能通过 Peer 节点发起交易请求。Peer 节点加入通道的时序图如图 10-6 所示。

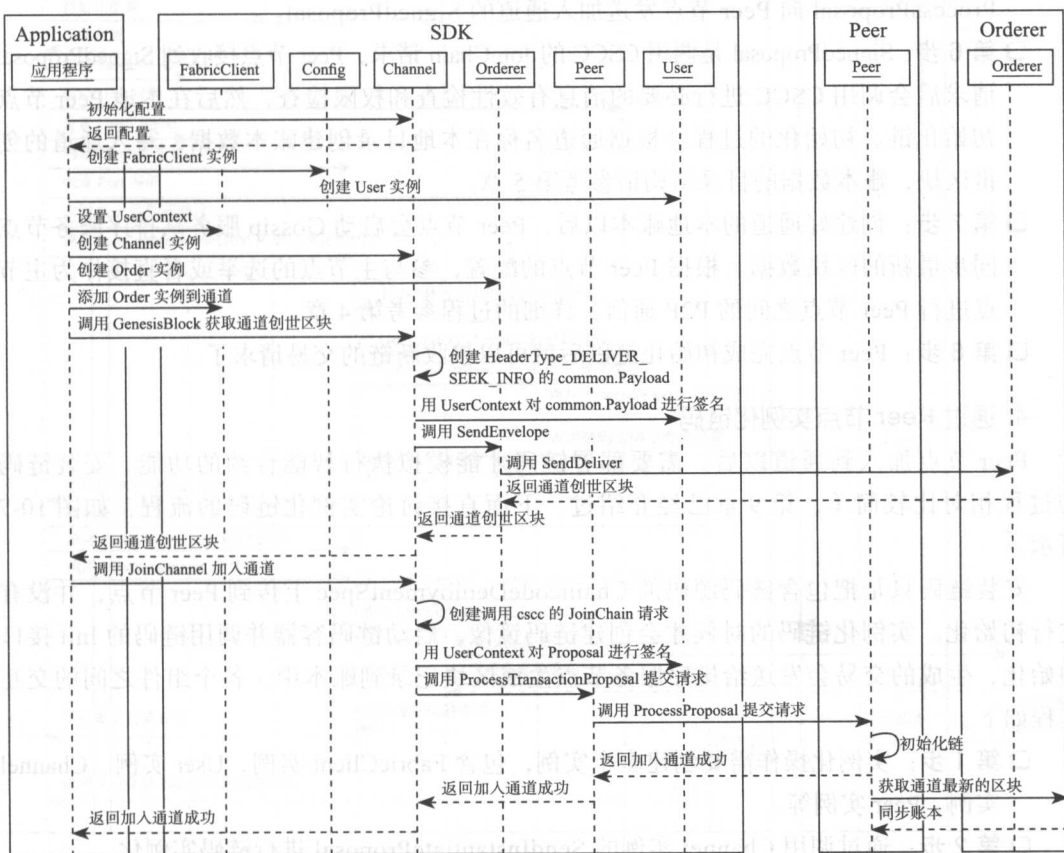


图 10-6 加入通道应用场景下 SDK 和其他组件交互的时序图

Peer 节点加入通道时需要先从排序服务节点获取创世区块，再在本地 Peer 节点初始化链，各个组件之间的交互如下。

- ❑ 第 1 步：必要的初始化配置，比如创建 FabricClient 实例，设置发起加入通道请求的用户、Channel 实例、Orderer 实例等。
- ❑ 第 2 步：调用 GenesisBlock 的请求获取创世区块，Channel 实例会构造 HeaderType\_DELIVER\_SEEK\_INFO 的请求，通过 Orderer 实例发送 SendDeliver 请求给排序服务节点，获取该通道的创世区块。
- ❑ 第 3 步：应用程序利用获取到的创世区块构造 JoinChannelRequest 请求，通过 Channel 实例发起 JoinChannel 请求。
- ❑ 第 4 步：HFC SDK 的 JoinChannel 操作会根据 JoinChannelRequest 请求重新构造类型为 HeaderType\_ENDORSER\_TRANSACTION 的 Proposal，Proposal 会用 FabricClient 实例设

置的用户进行签名,生成 SignedProposal。

- 第 5 步: 需要为每个加入通道的 Peer 节点创建一个 Peer 实例,通过 Peer 实例调用 ProcessProposal 向 Peer 节点发送加入通道的 SignedProposal。
- 第 6 步: SignedProposal 是调用 CSCC 的 JoinChain 请求,Peer 节点接收到 SignedProposal 请求后会调用 CSCC 进行必要的消息有效性检查和权限检查,然后在本地 Peer 节点初始化链。初始化的过程会根据通道名称在本地目录创建账本数据,写入通道的创世区块,账本数据的目录结构请参考第 5 章。
- 第 7 步: 创建好通道的本地账本以后,Peer 节点会启动 Gossip 服务从排序服务节点同步最新的区块数据。根据 Peer 节点的配置,参与主节点的选举或者直接作为主节点进行 Peer 节点之间的 P2P 通信,详细的过程参考第 4 章。
- 第 8 步: Peer 节点完成初始化链以后就可以接收新链的交易请求了。

#### 4. 通过 Peer 节点实例化链码

Peer 节点加入到通道以后,需要部署链码才能模拟执行智能合约的功能。安装链码的过程相对比较简单,第 9 章已经介绍过。下面直接讨论实例化链码的流程,如图 10-7 所示。

安装链码只是把包含链码源码的 ChaincodeDeploymentSpec 上传到 Peer 节点,并没有进行初始化。实例化链码的时候才会创建链码镜像,启动链码容器并调用链码的 Init 接口初始化,生成的交易会发送给排序服务节点生成区块记录到账本中。各个组件之间的交互过程如下。

- 第 1 步: 实例化操作需要创建多个实例,包含 FabricClient 实例、User 实例、Channel 实例、Peer 实例等。
- 第 2 步: 通过调用 Channel 实例的 SendInstantiateProposal 进行链码实例化。
- 第 3 步: HFC SDK 会构造包含 ChaincodeDeploymentSpec 的 ChaincodeInvocationSpec,调用的是 LSCC 的 deploy 请求,详细的结构请参考第 9 章。
- 第 4 步: 发送给 Peer 节点的请求同样会用 Channel 关联的用户进行签名,通过 Peer 实例的 ProcessTransactionProposal 提交生成的 SignedProposal。
- 第 5 步: 每次给 Peer 节点发送 SignedProposal 的时候都会新建一个 gRPC 的连接,通过 ProcessProposal 接口提交请求。
- 第 6 步: Peer 节点通过 SignedProposal 进行验证以后,会调用 LSCC 执行链码部署的操作,详细的过程请参考第 9 章。
- 第 7 步: Peer 节点返回的只是背书节点模拟执行和背书签名的结果,还需要提交给排序服务节点生成最终的区块才能生效,调用的过程同 Peer 节点接入通道的过程。
- 第 8 步: 生成的新区块会通过主节点分发给组织内的其他 Peer 节点,详细的内容请参考第 4 章。

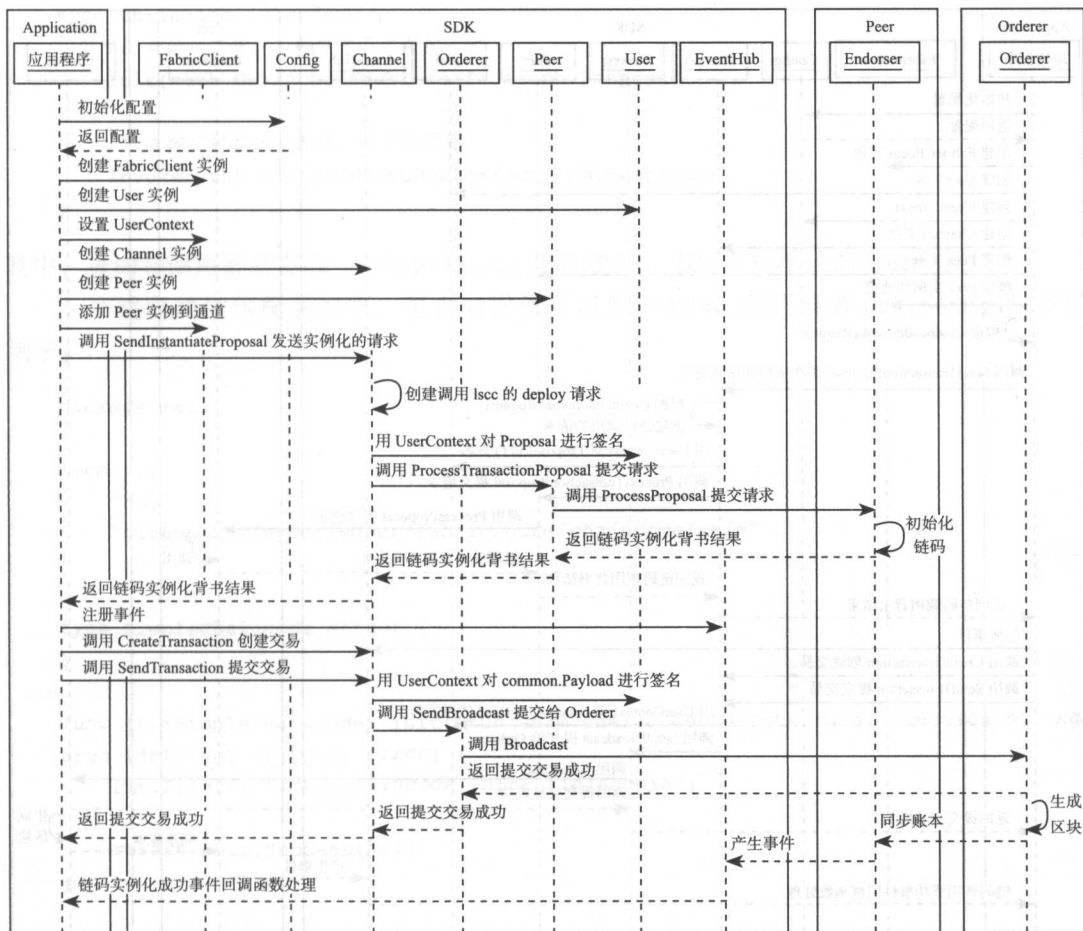


图 10-7 实例化链码应用场景下 SDK 和其他组件交互的时序图

## 5. 发起交易请求并生成区块

实例化链码的过程本身也是一种交易，所以发起交易请求的过程和上一节的过程非常类似，如图 10-8 所示。

这里只讨论和上一节实例化链码过程不同的几个地方。

- 普通的交易请求调用链码的 Invoke 接口，实例化链码调用的是 Init 接口。
- 普通的交易请求是不嵌套的 ChaincodeInvocationSpec 请求，包含通道的名称和调用链码的函数和参数等。
- 实例化链码的时候才开始构建链码镜像并启动链码容器，所以实例化链码的过程都比较慢。调用链码的背书节点已经启动了链码容器，所以调用链码的过程是比较快的，除非链码的功能比较复杂或者出于系统的原因。第一次接收调用链码请求的背书节点会自动构建链码镜像并启动链码容器，返回结果的时间会比较长。



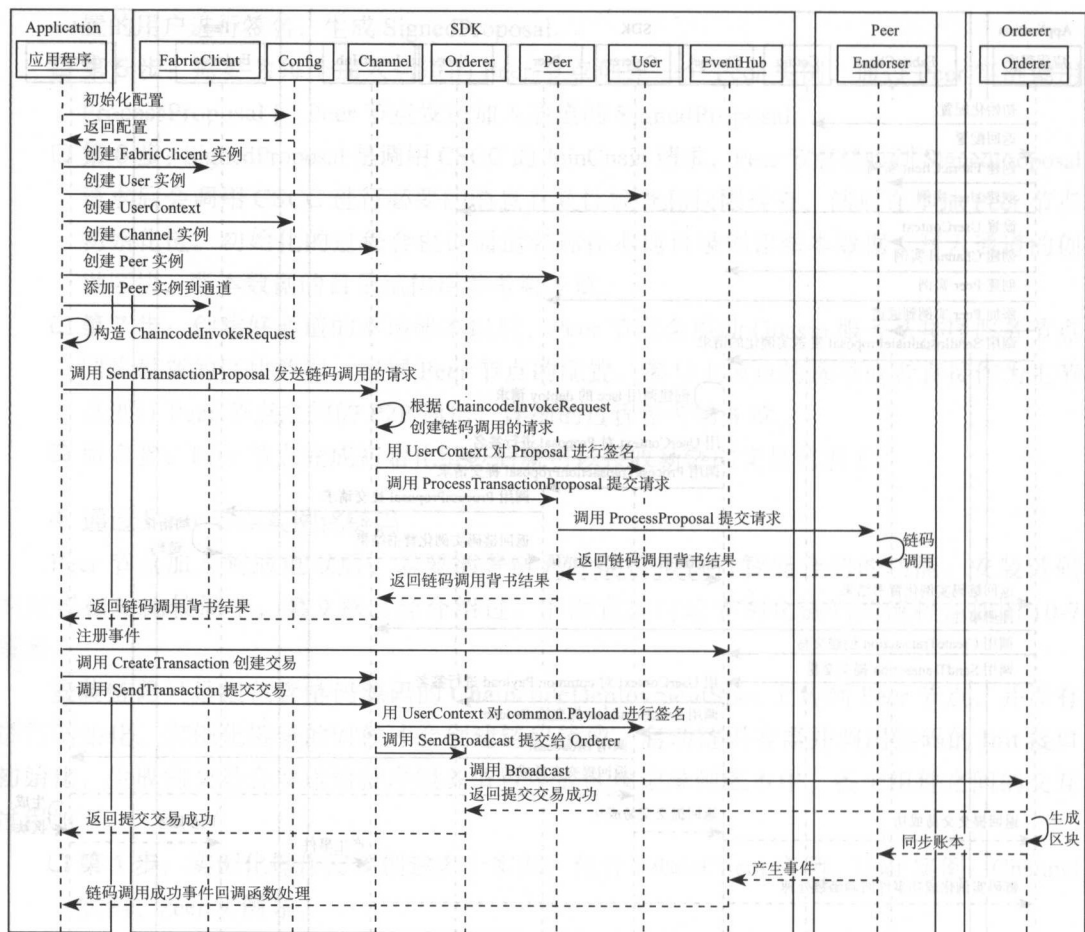


图 10-8 调用链码应用场景下 SDK 和其他组件交互的时序图

## 10.3 链码的开发和调试

链码是通过 SDK 和背书节点通信的，就是说链码的 SDK 只需要实现接口的定义就能和背书节点交互。理论上，链码是可以支持多种语言的。目前的版本（1.0.0）支持的语言只有 Golang，其他语言（比如 Java）还不够完善，正式发布的时候是禁用的。Car 支持利用 fabric-chaintool 打包的代码，目前也只支持 Golang。下面详细地介绍 Golang 语言的链码提供的接口，讨论如何开发和测试链码。

### 10.3.1 链码需要实现的接口

链码必须要实现的接口如下：



```

type Chaincode interface {
    // 初始化工作，一般情况下仅调用一次
    Init(stub ChaincodeStubInterface) pb.Response

    // 查询或更新状态数据，可多次调用
    Invoke(stub ChaincodeStubInterface) pb.Response
}

```

其中，查询和调用都是在同一个接口 `Invoke` 里实现的，不再实现单独的 `Query` 接口。

一个链码的样例模版如下，更多的样例可以参考 `fabric` 源码或者 `fabric-samples` 下的例子：

```

package main

import (
    "fmt"
    "github.com/hyperledger/fabric/core/chaincode/shim"
)

type SimpleChaincode struct {
}

func (t *SimpleChaincode) Init(stub shim.ChaincodeStubInterface, function string,
args []string) ([]byte, error) {
    fmt.Printf("SimpleChaincode Init: finished\n")

    return shim.Success(nil)
}

func (t *SimpleChaincode) Invoke(stub shim.ChaincodeStubInterface, function
string, args []string) ([]byte, error) {
    var err error
    func, args := stub.GetFunctionAndParameters()

    fmt.Printf("SimpleChaincode Invoke: func=%v, args = %v\n", func, args)

    return shim.Success(nil)
}

func main() {
    err := shim.Start(new(SimpleChaincode))
    if err != nil {
        fmt.Printf("Error starting Simple chaincode: %s", err)
    }
}

```

应用程序发起的调用是如何调用这两个接口的呢？

10.3.2 链码的 SDK 提供给链码的接口

链码的 SDK 是 shim，提供给链码的接口包括如下几种类型：

- ❑ 与链码调用参数解析相关；
- ❑ 与交易信息解析相关；
- ❑ 与状态数据操作相关；
- ❑ 与链码调用相关；
- ❑ 与事件处理相关；
- ❑ 与辅助操作相关。

下面逐一介绍每种类型具体的接口定义及其功能，这在实际的链码编写过程中都会用到。

1. 与链码调用参数解析相关的接口

shim 提供了多种获取链码调用参数的接口，可以根据不同的场景使用，如表 10-10 所示。

表 10-10 与链码调用参数解析相关的接口

接口名称	说明
GetArgs() byte	返回调用函数名称和参数的列表，第一个元素是函数名称，类型是字节数组
GetStringArgs() []string	返回调用函数名称和参数的列表，第一个元素是函数名称，类型是字符串
GetFunctionAndParameters() (string, []string)	分别返回调用的函数名称和参数，类型是字符串
GetArgsSlice() ([]byte, error)	返回调用函数名称和参数拼接在一起的字符数组

其中，接口 GetArgsSlice 返回的字符数组中间没有分隔符，没有办法还原成原始的函数名称和参数信息，实际链码编写过程中基本不会使用这个接口。

2. 与交易信息解析相关的接口

与提交的交易信息解析相关的接口如表 10-11 所示。

表 10-11 与交易信息解析相关的接口

接口名称	说明
GetTxID() string	获取交易号，每次调用这个值都是唯一的
GetCreator() ([]byte, error)	获取提交交易的身份信息（包括 MSP 和证书信息）
GetTransient() (map[string][]byte, error)	获取一些私密信息，这部分信息不会写入账本数据
GetBinding() ([]byte, error)	获取交易的绑定信息，包含随机数和提交交易的身份信息等
GetSignedProposal() (*pb.SignedProposal, error)	获取签名的 Proposal，签名者是和提交交易的身份一样的
GetTxTimestamp() (*timestamp.Timestamp, error)	获取提交交易时的时间戳，基于 UTC

需要特别说明的是 GetBinding 接口，binding 是和交易相关的一些指纹信息，计算方法如下：

```
binding = HASH(Nonce, Creator, Epoch)
```

其中，Epoch 代表的是时间窗口，背书节点在验证 Proposal 或者记账节点验证交易时会确认：

- 交易里的 Epoch 和当前 Epoch 的一致性；
- 同一个 Epoch 里没有出现相同交易号的交易。

目前 Epoch 的管理并没有实现，强制设置为 0。其他接口的说明参考前面 LSCC 里的内容。

### 3. 与状态数据操作相关的接口

与状态数据操作相关的接口又分为 3 类：

- 基于单一键操作的接口；
- 基于键范围查询的接口；
- 基于值内容查询的接口。

链码基于单一键操作的接口如表 10-12 所示。

表 10-12 链码基于单一键操作的接口

接口名称	说明
GetState(key string) ([]byte, error)	根据指定键查询状态数据库里存储的值
PutState(key string, value []byte) error	向状态数据库写入键值对（模拟写入，记账的时候才写）
DelState(key string) error	删除状态数据库里键对应的值（模拟删除，记账的时候才删）
GetHistoryForKey(key string) (HistoryQueryIteratorInterface, error)	查询一个键的历史数据

链码基于键范围查询的接口如表 10-13 所示。

表 10-13 链码基于键范围查询的接口

接口名称	说明
GetStateByRange(startKey, endKey string) (StateQueryIteratorInterface, error)	查询状态数据库里键在 [startKey, endKey) 之间的值，包含 startKey，不包含 endKey
CreateCompositeKey(objectType string, attributes []string) (string, error)	构造组合键
SplitCompositeKey(compositeKey string) (string, []string, error)	分割组合键
GetStateByPartialCompositeKey(objectType string, keys []string) (StateQueryIteratorInterface, error)	部分组合键查询

部分组合键查询实际按键的前缀查询，例如，某个链码的组合键格式为 K1-K2-K3，支持按 K1 或 K1-K2 前缀查询，底层转化为区间查询实现。更多组合键查询的内容参考 5.6 节。

链码基于值内容查询的接口如表 10-14 所示。

表 10-14 链码基于内容查询的接口

接口名称	说明
GetQueryResult(query string) (StateQueryIteratorInterface, error)	根据指定条件查询状态数据库里存储的值

基于值的查询需要状态数据库的支持，目前只有 CouchDB 才能支持。

4. 与链码调用相关的接口

与链码调用相关的接口如表 10-15 所示。

表 10-15 与链码相互调用的接口

接口名称	说明
InvokeChaincode(chaincodeName string, args byte, channel string) pb.Response	根据指定条件查询状态数据库里存储的值

链码是可以调用其他链码的接口的，目前实现的原则如下。

- ❑ 如果被调用的链码和调用的链码在相同的链上，会继承交易模拟器，被调用链码生成的读写集添加到交易中，读写集是以链码编号（chaincodeId）作为命名空间的。就是说相同链的链码调用的读写都是有效的，会影响到被调用链码的状态数据库。
- ❑ 如果被调用的链码和调用的链码不在相同的链上，会生成一个新的交易模拟器，返回调用执行的 Response 给链码，不会添加读写集到交易中。就是说，不同链的链码调用只能查询状态数据，不能写入。

5. 与事件处理相关的接口

与事件处理相关的接口如表 10-16 所示。

表 10-16 与链码事件处理相关的接口

接口名称	说明
SetEvent(name string, payload []byte) error	设置事件的名称和内容

6. 与辅助操作相关的接口

与辅助操作相关的接口如表 10-17 所示。

表 10-17 与链码辅助操作相关的接口

接口名称	说明
Start(cc Chaincode) error	启动链码
NewLogger(name string) *ChaincodeLogger	返回日志操作的对象，每条日志都会有 name 的标识
SetLevel(level LoggingLevel)	设置日志的级别

日志操作相关的接口：Debug、Info、Notice、Warning、Error、Critical 和 Debugf、Infof、Noticef、Warningf、Errorf、Criticalf 等，其中带 'f' 的版本是有格式化字符串的。

链码的 SDK 也会有日志，名称是 shim，日志级别由环境变量 CORE\_CHAINCODE\_LOGGING\_SHIM 控制，是启动链码的时候传递给链码容器的。

10.3.3 链码开发的注意事项

目前链码的设计并没有考虑区块链外部数据源的问题，在链码开发过程中需要注意以下的事项。

- ❑ 不能使用不确定性的变量作为计算的输入：比如不能采用随机数或者获取系统当前时间等。链码会在不同的节点上多次运行，但是运行的时间并不严格一致，环境的查询导致不同节点上执行的结果不一致，最终无法达成共识。
- ❑ 避免调用外部数据接口导致重复计算：比如多个节点多次调用外部写数据的接口，可能会导致区块链外部重复计算。这种情况多个链码的执行结果可能是一致的，不会导致共识失败，但会影响外部的一致性，这是一种逻辑错误。

### 10.3.4 链码的调试

自动部署的链码都是运行在容器中的，开发调试过程比较复杂，需要经历如下的步骤，如图 10-9 所示。

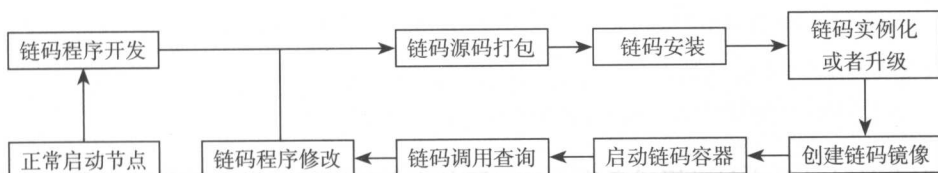


图 10-9 生产环境下链码的调试流程图

其实，Fabric 本身提供了开发模式启动节点：

```
peer node start --peer-chaincodedev=true
```

省略中间的创建通道的过程，编译和启动链码的过程如下：

```
cd examples/chaincode/go/chaincode_example02
go build
CORE_CHAINCODE_LOGLEVEL=debug CORE_PEER_ADDRESS=127.0.0.1:7051 CORE_CHAINCODE_ID_
NAME=mycc:0 ./chaincode_example02
```

后面是正常的安装和实例化的操作，步骤如图 10-10 所示。

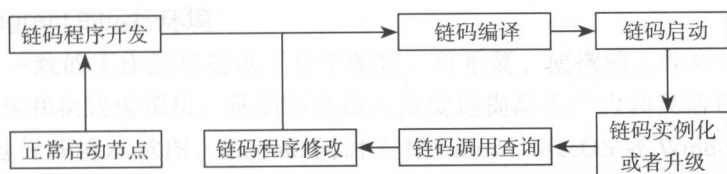


图 10-10 调试环境下链码的调试流程图

在这个模式下，链码直接以可执行的程序运行，开发调试不再需要创建链码镜像，能够快速调试链码程序。

## 10.4 本章小结

本章先介绍了基于超级账本的应用开发模型，从应用开发的角度看，主要有两个部分，一个部分是基于不同语言的 SDK 开发和区块链网络交互的应用程序，另外一个部分是实现超级账本的智能合约。目前 SDK 提供了 4 种语言的版本，本章主要以 Golang 版本为基础介绍了 SDK 的各个模块及其主要的功能，然后介绍了几种主要应用场景的调用时序图，方便了解应用程序、SDK 和超级账本节点之间的交互关系。

最后介绍了超级账本的智能合约，链码的基本框架，链码的主要接口及其功能，如何编写链码和调试。

了解了以上的内容，就能够动手搭建符合自身业务场景的区块链网络，利用区块链技术实现具有智能合约、不可篡改等特性的应用。



图 10-1 超级账本应用开发模型

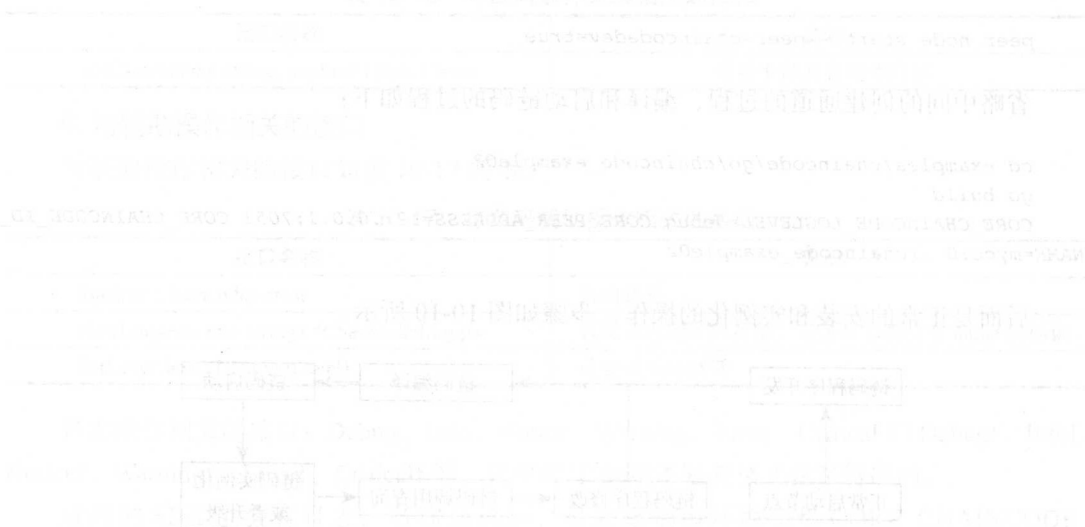


图 10-2 应用开发模型调用时序图

## 从零开始部署超级账本网络

在第 2 章我们简要地搭建了超级账本的网络，本章我们详细地介绍如何从头进行初始化的配置，手动部署超级账本的节点和链码，调用链码实现智能合约的功能。

### 11.1 准备超级账本运行环境

本节介绍多种构建超级账本运行环境的方法，然后介绍如何编译超级账本的镜像文件。

#### 11.1.1 超级账本运行环境

链码依赖于 Docker 才能启动运行，超级账本的各节点也推荐运行在 Docker 容器中，方便系统的运维管理。在开发的过程中，有多种运行方式可以选择：基于 Vagrant 的运行环境、基于 Virtualbox 的运行环境和基于 Docker 的运行环境。

##### 1. 基于 Vagrant 的运行环境

Vagrant 用一致的工作流程提供了易于配置、可重复、便携的工作环境，让开发人员可以快速地创建和销毁虚拟机，帮助团队最大限度地提高生产力和灵活性。图 11-1 是基于 Vagrant 的运行环境示意图，Vagrant 同时支持 Linux、MacOS 或 Windows 等不同的平台，通过相同的配置文件 Vagrantfile 启动 Virtualbox 虚拟机。虚拟机的操作系统是 Ubuntu 16.04，启动的过程会调用脚本 `devenv/setup.sh` 安装所需要的软件并做相应配置，包括安装 Docker、Docker Compose、Golang、Node.js、OpenJDK、Gradle 等，还会修改系统的文件描述符数等，这样统一用同一套配置运行相同的程序，就不会再有“我的环境有问题”的借口了，也不会存在工作习惯划分出不同阵营的情况，非常适合工作团队合作。图 11-1 是



基于 Vagrant 的运行环境示意图。

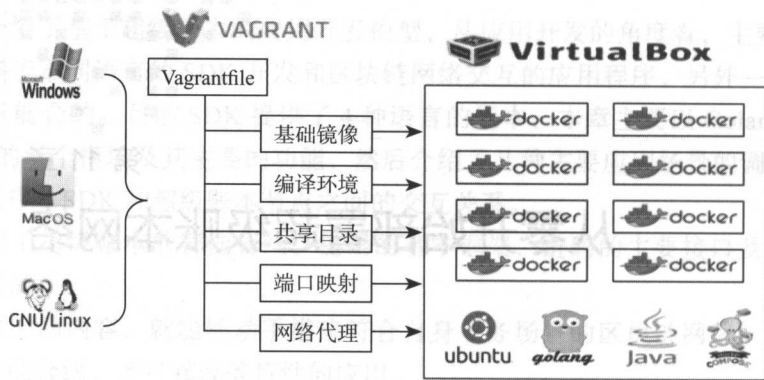


图 11-1 基于 Vagrant 的运行环境示意图

### (1) 配置文件 Vagrantfile

我们先来看一下配置文件 Vagrantfile 的内容。

```
SRCMOUNT = "/hyperledger"
LOCALDEV = "/local-dev"

$script = <<SCRIPT
set -x
echo "127.0.0.1 couchdb" | tee -a /etc/hosts
export DOCKER_STORAGE_BACKEND="#{ENV['DOCKER_STORAGE_BACKEND']}"
cd //{SRCMOUNT}/devenv
./setup.sh
SCRIPT

Vagrant.require_version ">= 1.7.4"
Vagrant.configure('2') do |config|
  config.vm.box = "ubuntu/xenial64" // 基础镜像
  config.vm.network :forwarded_port, guest: 7050, host: 7050, id: "orderer",
  host_ip: "localhost", auto_correct: true // 排序服务
  config.vm.network :forwarded_port, guest: 7051, host: 7051, id: "peer", host_ip:
  "localhost", auto_correct: true // Peer 节点
  config.vm.network :forwarded_port, guest: 7053, host: 7053, id: "peer_event",
  host_ip: "localhost", auto_correct: true // 事件服务
  config.vm.network :forwarded_port, guest: 7054, host: 7054, id: "ca", host_ip:
  "localhost", auto_correct: true // fabric-ca
  config.vm.network :forwarded_port, guest: 5984, host: 15984, id: "couchdb",
  host_ip: "localhost", auto_correct: true // CouchDB
  config.vm.synced_folder "..", "#{SRCMOUNT}" // 同步目录
  config.vm.synced_folder "..", "/opt/gopath/src/github.com/hyperledger/fabric"
  // 同步目录
  config.vm.synced_folder ENV.fetch('LOCALDEVDIR', ".."), "#{LOCALDEV}" // 同步目录
  if File.exist?("../..//fabric-ca")
    config.vm.synced_folder "../..//fabric-ca", "/opt/gopath/src/github.com/
```

```

hyperledger/ fabric-ca"
end
config.vm.provider :virtualbox do |vb|
  vb.name = "hyperledger" // 虚拟机名称
  vb.customize ['modifyvm', :id, '--memory', '4096'] // 内存大小
  vb.cpus = 2 // CPU
  storage_backend = ENV['DOCKER_STORAGE_BACKEND']
  case storage_backend
  when nil, "", "aufs", "AUFS"
    // 不需要额外的操作
  when "btrfs", "BTRFS"
    // 添加一个 btrfs 的卷
    IO.popen("VBoxManage list systemproperties") { |f|
      success = false
      while line = f.gets do
        // 查找虚拟机文件存储目录
        machine_folder = line.sub(/^Default machine folder: s*/, "")
        if line != machine_folder
          btrfs_disk = File.join(machine_folder, vb.name, 'btrfs.vdi')
          unless File.exist?(btrfs_disk)
            // 创建 btrfs 磁盘
            vb.customize ['createhd', '--filename', btrfs_disk,
              '--format', 'VDI', '--size', 20 * 1024]
          end
          // 添加磁盘到虚拟机
          vb.customize ['storageattach', :id, '--storagectl', 'SATA
            Controller', '--port', 1, '--device', 0, '--type', 'hdd',
            '--medium', btrfs_disk]
          success = true
          break
        end
      end
      raise Vagrant::Errors::VagrantError.new, "Could not provision btrfs
        disk" if !success
    }
  else
    raise Vagrant::Errors::VagrantError.new, "Unknown storage backend type:
      {storage_backend}"
  end
end
end
config.vm.provision :shell, inline: $script
end

```

Vagrant.require\_version “>= 1.7.4” 是对 Vagrant 本身版本的要求，这里定义的版本必须大于等于 1.7.4。config.vm.box 定义了基础镜像的名称和版本，config.vm.network 定义了与宿主机之间的端口转发，预留端口间的对应关系如表 11-1 所示。实际用途可以自己

表 11-1 Vagrant 的端口映射列表

宿主机端口	虚拟端口	说明
7050	7050	排序服务端口
7051	7051	Peer 节点的 gRPC 服务端口
7053	7053	事件服务端口
7054	7054	fabric-ca 服务端口
15984	5984	CouchDB 服务端口

定义，也可以修改这个文件增加端口的映射。

`config.vm.synced_folder` 是和宿主机共享目录的配置，默认是把源代码的目录映射到了 `/hyperledger`、`/local-dev`、`/opt/gopath/src/github.com/hyperledger/fabric` 等几个目录下，任何一个目录下修改都能在其他目录下看到变化，和宿主机上的目录是同步的。

## (2) Vagrant 镜像文件 Box

Vagrant 生成的镜像叫 Box，官方提供了存储仓库：<https://atlas.hashicorp.com/boxes/search>，可以搜索到公开的 Box，比如 `hyperldger` 的基础镜像 <https://atlas.hashicorp.com/hyperledger/boxes/fabric-baseimage>，最新版本是 0.3.0。目前官方并不推荐使用 Vagrant 的方式构建开发环境，已经很久没有更新了。

Box 支持 Docker、Hyper-V、VMware、VirtualBox 等不同的 Provider 类型。Box 是采用 `tar`、`tar.gz` 或者 `zip` 压缩的，压缩包里的内容根据 Provider 不同有所区别，例如，本地的一个 `baseimage-public.box` 里包含内容如下。

```
localhost:fabric-baseimage clarity$ tar -tf baseimage-public.box
Vagrantfile
box.ovf
metadata.json
packer-virtualbox-iso-1482837643-disk1.vmdk
```

其中 `metadata.json` 是必须的，需要指定 Provider 的类型，内容如下：

```
localhost:fabric-baseimage clarity$ cat metadata.json
{"provider":"virtualbox"}
```

VirtualBox 的虚拟机默认使用第一个网卡进行通信，需要设置成 NAT 模式，Vagrantfile 里的 `config.vm.base_mac` 设置的就是 NAT 网络设备的 macOS 地址：

```
localhost:fabric-baseimage clarity$ cat Vagrantfile
Vagrant.configure("2") do |config|
  config.vm.base_mac = "08002730B696"
end
```

Vagrant 启动 VirtualBox 虚拟机的时候会导入压缩包里的 `box.ovf` 文件，OVF 是 Open Virtualization Format 的简称，是一种开放的虚拟机打包和分发的标准，更多的内容参考 <http://www.dmtf.org/standards/ovf>。根据“References->File 的 `ovf:href`”读取真正的虚拟机文件，其他是 CPU、网络、内存等的设置。

## (3) Vagrant 支持的虚拟机环境

Provider 是 Vagrant 支持的虚拟机运行环境，目前支持 Docker、Hyper-V、VMware、VirtualBox 等几种类型，是和 Box 的文件格式相对应的。除此之外，还可以对 Provider 进行定制，<https://github.com/mitchellh/vagrant-aws> 以插件的形式提供了对 AWS 的支持，也可以用 Ruby 语言编写自己的插件。

## (4) Vagrant 常用命令 (见表 11-2)

表 11-2 Vagrant 常用命令

命令	说明
vagrant box list	镜像查看
vagrant box add ADDRESS	增加名称为 ADDRESS 的镜像
vagrant up [name or id]	根据 Vagrantfile 文件启动镜像
vagrant reload	在 Vagrantfile 修改了以后, 可以通过重启生效, 相当于执行了 vagrant halt && vagrant up
vagrant ssh	ssh 登录到虚拟机
vagrant status	查看虚拟机运行状态

更多的命令可以查看官方文档, 或者通过参数 -h 查看:

```
vagrant -h
```

## (5) Vagrant 的安装和使用

安装过程比较简单, MacOS、Windows、Debian、CentOS 等平台都有安装包, 直接到下载页面 <https://www.vagrantup.com/downloads.html> 下载最新版本安装即可。提醒一下, 在实际的测试过程中发现 Vagrant 和 VirtualBox 也存在版本依赖的情况, 比如 Vagrant 1.8.4 和 Virtualbox 的 5.1.X 的版本会有冲突, 出现 “No usable default provider could be found for your system.” 的问题。安装的时候不要下载和 Virtualbox 有冲突的版本就可以。

安装完成以后进入超级账本源码子目录 devenv, 执行如下命令就可以自动配置好开发环境了:

```
// 进入超级账本源码子目录 devenv
cd $GOPATH/src/github.com/hyperledger/fabric/devenv/
// 启动 Vagrant 环境, 自动配置开发环境。如果 vagrant 环境已经启动, 则不需要执行
vagrant up
// 进入虚拟机环境
vagrant ssh
```

配置 Vagrant 环境的过程可能会比较慢, 可能会有一些网络方面的问题, 可以参考本章后面要介绍的编译镜像文件部分做修改。配置完成以后就安装好了 Golang、Docker、Docker Compose 等环境。准备好开发环境就可以进行源码编译了, 参考本章后面编译镜像文件的内容。

## 2. 基于 Virtualbox 的运行环境

基于 Vagrant 的运行环境最大的好处是能自动地配置开发环境, 不过 Vagrant 也会存在和操作系统兼容性的问题, 比如在 Windows 7 上很容易安装失败, 最新版本中超级账本官方并不推荐采用 Vagrant 的方式。我们可以在虚拟机的基础上自行构建所需要的开发环境, 这里我们以 Virtualbox 为例。

VirtualBox 也支持 Windows、MacOS、Linux、Solaris 等多个平台，直接在官网 <https://www.virtualbox.org/wiki/Downloads> 下载安装即可。这里简单地介绍一下在 Virtualbox 上安装 Ubuntu 16.04 的过程，先从国内的镜像源下载镜像文件：<http://mirrors.163.com/ubuntu-releases/16.04/ubuntu-16.04.3-server-amd64.iso>。

在打开的 VirtualBox 控制台上点击新建虚拟机，在操作系统类型中选择 Ubuntu，虚拟机的名称可以自定义，比如 Ubuntu，如图 11-2 所示，然后按照默认选项点击下一步的按钮。

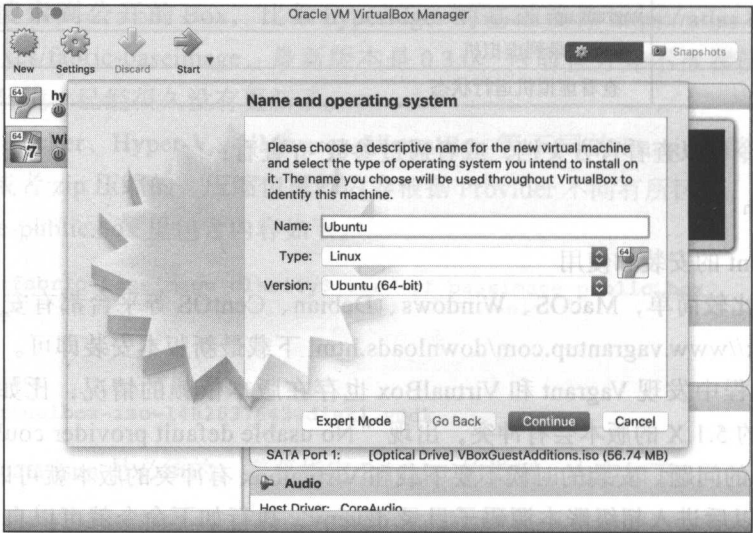


图 11-2 在 Virtualbox 上新建虚拟机

当虚拟机新建完成以后，在虚拟机列表中选择创建的虚拟机，如图 11-3 所示。

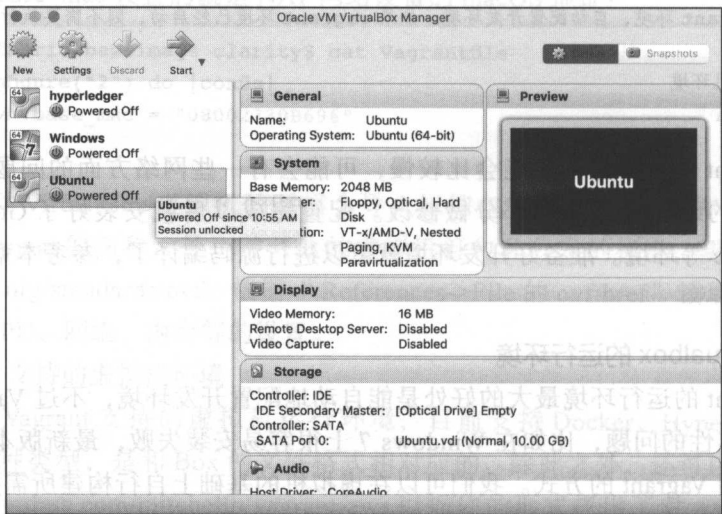


图 11-3 启动虚拟机

这个时候虚拟机是没有安装操作系统的，在启动过程中可以选择刚刚下载的 iso 文件，如图 11-4 所示。

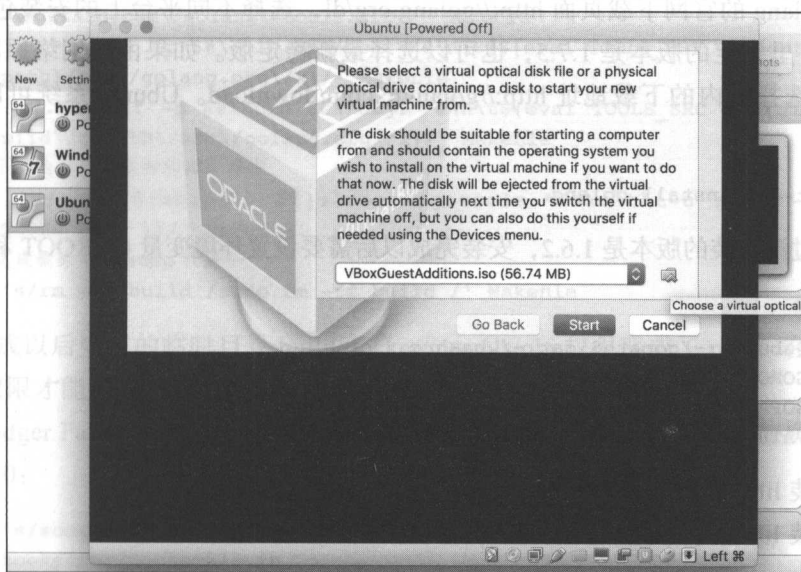


图 11-4 选择操作系统安装文件

在弹出的文件浏览器中选择文件，按默认步骤安装就可以了，如图 11-5 所示。

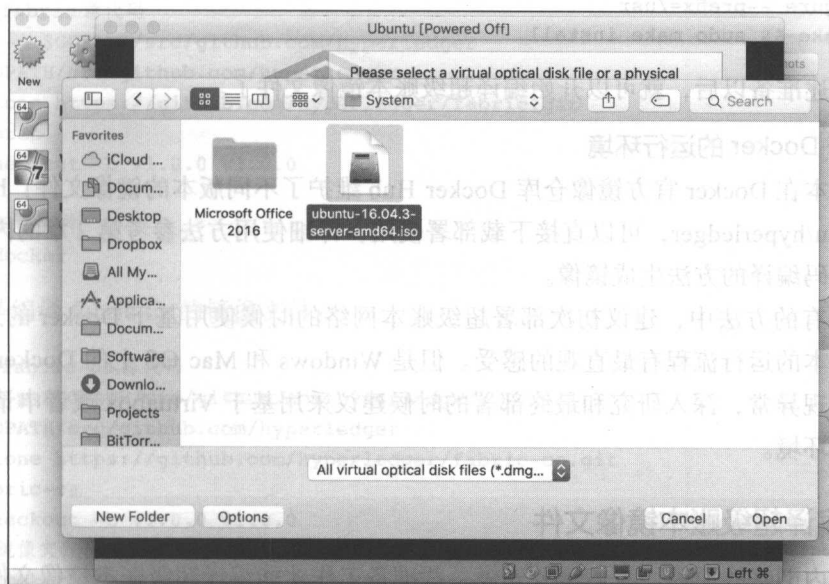


图 11-5 选择操作系统安装文件

当操作系统安装完成以后就可以安装一些基础环境了，Docker 和 Docker Compose 的安



装请参考第 2 章的内容。

### (1) 安装 Golang

进入 Golang 的官网下载页面 <http://golang.org/dl>, 选择不同平台上的安装包安装即可, 目前 Vagrant 里指定的版本是 1.7.5, 也可以选择最新稳定版。如果由于网络原因无法下载的话, 可以选择国内的下载地址 <http://golangtc.com/download>。Ubuntu 系统可以直接通过命令安装:

```
sudo apt-get install golang
```

直接通过源安装的版本是 1.6.2, 安装完成以后需要设置环境变量 GOROOT 和 GOPATH, 比如:

```
clarity@ubuntu:~/gopath$ cat ~/.bashrc
export GOROOT=/usr/local/go
export GOPATH=$HOME/gopath
export PATH=$PATH:$GOROOT/bin:$GOPATH/bin
```

### (2) 安装 libtool

需要安装 libtool, 否则后面编译的时候会报错: 'ltdl.h' file not found。

```
wget http://ftpmirror.gnu.org/libtool/libtool-2.4.6.tar.gz
tar -zxvf libtool-2.4.6.tar.gz
cd libtool-2.4.6
sudo apt-get install automake
./configure --prefix=/usr
sudo make && sudo make install
```

做了上述准备以后, 就可以开始编译超级账本镜像文件了。

## 3. 基于 Docker 的运行环境

超级账本在 Docker 官方镜像仓库 Docker Hub 维护了不同版本的镜像文件: <https://hub.docker.com/u/hyperledger>, 可以直接下载部署使用, 详细使用方法参考第 2 章的内容。下面我们介绍源码编译的方法生成镜像。

上述所有的方法中, 建议初次部署超级账本网络的时候使用基于 Docker 的运行环境, 能对超级账本的运行流程有最直观的感受。但是 Windows 和 Mac OS 上的 Docker 都极不稳定, 容易出现异常, 深入研究和最终部署的时候建议采用基于 Virtualbox 或者申请云主机镜像搭建运行环境。

### 11.1.2 编译超级账本镜像文件

由于国内网络的原因, 我们需要先做一些准备工作才能编译超级账本镜像文件。

#### 1. 编译超级账本镜像文件

下面是通过 Github 下载 [golang.org/x/tools](https://github.com/golang/x/tools) 库的方法, 否则可能由于网络问题安装失败。



```
// 通过 Github 下载 golang.org/x/tools 库
sed -i 's/.*@mkdir -p $@/bin $@/obj.*/&\n\tcd $(TOOLS_SRC)/tools || (mkdir -p
$(TOOLS_SRC) \&\& cd $(TOOLS_SRC) \&\& git clone
https://github.com/golang/tools.git)/' Makefile
sed -i 's/.*@mkdir -p $@/bin $@/obj.*/&\n\t$(eval TOOLS_DST = /opt/gotools/
obj/gopath/src/golang.org/x)/' Makefile
sed -i 's/.*@mkdir -p $@/bin $@/obj.*/&\n\t$(eval TOOLS_SRC = $(dir $(abspath
$<))\build/gopath/src/golang.org/x)/' Makefile
// 映射下载的目录到临时构建容器中
sed -i 's/.*-v $(abspath $@):/opt/gotools.*/&\n\t-t-u root -v $(TOOLS_
SRC):$(TOOLS_DST) \\\/' Makefile
// 构建完成删除增加 sudo 权限
sed -i 's/rm -rf build /sudo rm -rf build /' Makefile
```

编译完成以后生成的临时目录 build/docker/gotools/obj 权限变成了 root，在 make clean 需要 sudo 权限才能删除。

Hyperledger Fabric 1.0.0 中 Zookeeper 的版本是 3.4.9，已经不维护了，替换为最新的稳定版本 3.4.10：

```
sed -i 's/zookeeper-3.4.9/zookeeper-3.4.10/'
images/zookeeper/Dockerfile.in
```

做好前面的准备工作后，超级账本源码编译就比较简单了，只需要在超级账本的源代码目录下执行如下命令，就可以自动编译出所有的镜像文件了。

```
# 下载 fabric 源代码
mkdir -p $GOPATH/src/github.com/hyperledger
cd $GOPATH/src/github.com/hyperledger
git clone https://github.com/hyperledger/fabric.git
cd fabric
git checkout -b v1.0.0 v1.0.0
```

```
# 编译镜像文件
make docker
```

下面是编译 fabric-ca 的镜像文件：

```
# 下载 fabric-ca 源代码
mkdir -p $GOPATH/src/github.com/hyperledger
cd $GOPATH/src/github.com/hyperledger
git clone https://github.com/hyperledger/fabric-ca.git
cd fabric-ca
git checkout -b v1.0.0 v1.0.0
# 编译镜像文件
make docker
```

## 2. 超级账本镜像文件

编译生成的超级账本镜像文件，如表 11-3 所示。

表 11-3 超级账本镜像文件

镜像名称	是否可选	镜像说明
hyperledger/fabric-tools	可选	包含 cryptogen、configtxgen、configtxlator 等工具的镜像文件
hyperledger/fabric-couchdb	可选	CouchDB 的数据库镜像文件，状态数据库选择 CouchDB 的时候才需要
hyperledger/fabric-kafka	可选	Kafka 的镜像文件
hyperledger/fabric-zookeeper	可选	Zookeeper 的镜像文件
hyperledger/fabric-orderer	必选	排序服务节点的镜像文件
hyperledger/fabric-peer	必选	Peer 节点的镜像文件
hyperledger/fabric-javaenv	可选	Java 链码的基础镜像文件
hyperledger/fabric-ccenv	必选	Golang 链码的基础镜像文件
hyperledger/fabric-ca	可选	fabric-ca 的镜像文件，用到 fabric-ca 的时候才需要

图 11-6 是超级账本的镜像文件的构建和运行关系图。

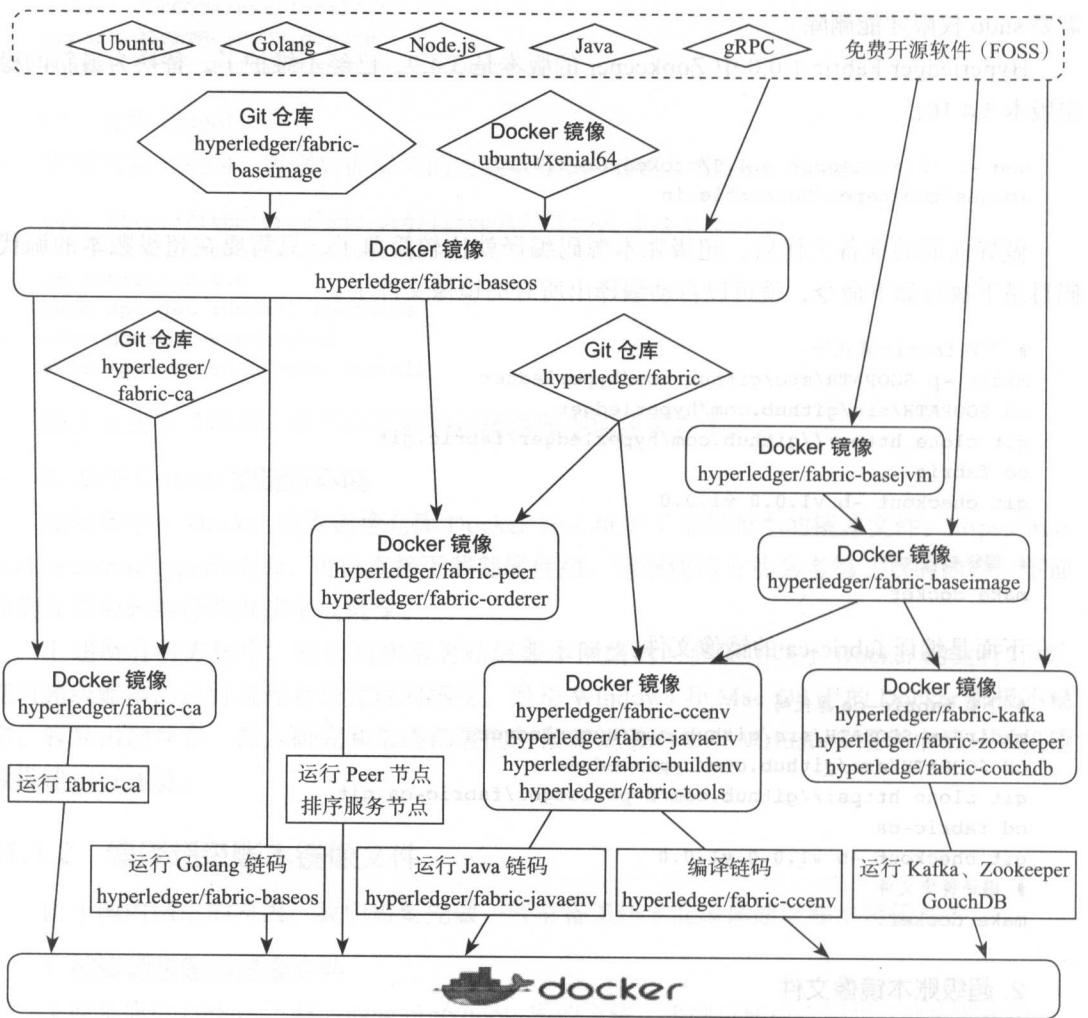


图 11-6 镜像文件的构建和运行关系图

编译超级账本镜像用到很多的免费开源软件 (Free and Open-source Software, FOSS), 基于 Ubuntu 16.04 的镜像文件 ubuntu/xenial64 构建出镜像文件 hyperledger/fabric-baseos, 这是所有其他镜像文件的基础。镜像文件较小, 包含的内容也只在 ubuntu/xenial64 基础上增加了软件下载工具 wget 和时区配置工具 tzdata。Golang 的链码是在 hyperledger/fabric-baseos 基础上包含通过 hyperledger/fabric-ccenv 编译出的二进制文件构建的链码容器, 这样构建出的链码容器相对较小。镜像文件 hyperledger/fabric-basejvm 包含了 OpenJDK 8 的 Java 编译运行环境, Java 的链码基于 hyperledger/fabric-javaenv 的运行环境, 包含了 Java 的链码 SDK 的内容。

## 11.2 快速构建超级账本网络

我们基于 fabric-samples 里的 BYFN (Build Your First Network) 介绍超级账本的构建过程, 首先是利用提供的脚本快速地构建网络, 后面是详细的构建过程。

### 11.2.1 下载 BYFN 的代码

BYFN 是包含在 fabric-samples 的 first-network 目录下的, 先通过 git 下载源代码:

```
cd $GOPATH/src/github.com/hyperledger
git clone https://github.com/hyperledger/fabric-samples.git
cd fabric-samples/first-network
```

后面的操作默认都在此路径下进行。

### 11.2.2 BYFN 脚本介绍

运行 BYFN 脚本需要已经安装好 Docker 基础环境, 编译出镜像文件和系统工具 cryptogen、configtxgen 等。BYFN 脚本的帮助说明如下:

```
claritys-MacBook-Pro:first-network clarity$ ./byfn.sh -h
// 用法:
byfn.sh -m up|down|restart|generate [-c <channel name>] [-t <timeout>]
byfn.sh -h|--help 显示帮助信息
-m <mode> - one of 'up', 'down', 'restart' or 'generate'
  - 'up' - 使用 docker-compose 启动网络
  - 'down' - 使用 docker-compose 停止网络
  - 'restart' - 重启网络
  - 'generate' - 生成证书和创世区块
-c <channel name> - 通道名称, 默认是 "mychannel"
-t <timeout> - 超时时间, 默认是 10000 毫秒
// 下面是通用的流程, 先生成证书和创世区块, 再启动网络:
byfn.sh -m generate -c <channelname>
byfn.sh -m up -c <channelname>
```

```
byfn.sh -m down -c <channelname>

// 默认选项:
byfn.sh -m generate
byfn.sh -m up
byfn.sh -m down
```

其中, '-c' 参数指定通道名称, 默认是 mychannel, '-t' 是连接的超时时间, 默认是 10 秒。下面简要介绍其中的几个功能, 如表 11-4 所示。

表 11-4 BYFN 脚本的命令选项

命令选项	功能说明
byfn.sh -m generate	根据配置文件 crypto-config.yaml 生成初始化配置, 包含 Peer 节点、排序服务节点的 MSP 证书, 根据 configtx.yaml 生成创世区块等
byfn.sh -m up	根据生成的初始化配置, 启动超级账本网络
byfn.sh -m down	停止超级账本网络

下面我们利用 BYFN 脚本构建出超级账本网络。

11.2.3 生成网络初始化配置

执行脚本 ./byfn.sh -m generate 生成 MSP 证书和创世区块:

```
./byfn.sh -m generate
```

执行结果如下:

```
Generating certs and genesis block for with channel 'mychannel' and CLI timeout
of '10000'
Continue (y/n)? y
proceeding ...
/Users/clarity/Projects/bin/cryptogen

#####
#####          用 cryptogen 工具生成证书          #####
#####
org1.example.com
org2.example.com

/Users/clarity/Projects/bin/configtxgen
#####
#####          生成排序服务的创世区块          #####
#####
2017-08-09 14:03:16.410 CST [common/configtx/tool] main -> INFO 001 Loading configuration
2017-08-09 14:03:16.437 CST [common/configtx/tool] doOutputBlock -> INFO 002 Generating
genesis block
2017-08-09 14:03:16.440 CST [common/configtx/tool] doOutputBlock -> INFO 003 Writing
genesis block
```

```
#####
#####      生成通道配置交易 'channel.tx'      #####
#####
2017-08-09 14:03:16.460 CST [common/configtx/tool] main -> INFO 001 Loading configuration
2017-08-09 14:03:16.464 CST [common/configtx/tool]
doOutputChannelCreateTx -> INFO 002 Generating new channel configtx
2017-08-09 14:03:16.464 CST [common/configtx/tool]
doOutputChannelCreateTx -> INFO 003 Writing new channel tx
```

```
#####
#####      生成 Org1MSP 锚节点配置      #####
#####
2017-08-09 14:03:16.488 CST [common/configtx/tool] main -> INFO 001 Loading configuration
2017-08-09 14:03:16.493 CST [common/configtx/tool] doOutputAnchorPeersUpdate -> INFO
002 Generating anchor peer update
2017-08-09 14:03:16.493 CST [common/configtx/tool] doOutputAnchorPeersUpdate -> INFO
003 Writing anchor peer update
```

```
#####
#####      生成 Org2MSP 锚节点配置      #####
#####
2017-08-09 14:03:16.525 CST [common/configtx/tool] main -> INFO 001 Loading configuration
2017-08-09 14:03:16.529 CST [common/configtx/tool] doOutputAnchorPeersUpdate -> INFO
002 Generating anchor peer update
2017-08-09 14:03:16.531 CST [common/configtx/tool] doOutputAnchorPeersUpdate -> INFO
003 Writing anchor peer update
```

执行成功以后生成的 Peer 节点和排序服务节点的 MSP 证书文件在目录 `crypto-config` 下, 包含 4 个 Peer 节点 (分成两个组织, 即 `org1.example.com` 和 `org2.example.com`) 和 1 个排序服务节点, 排序后端默认使用 SOLO, 详细的目录结构参考第 8 章的内容。

生成的创世区块在目录 `channel-artifacts` 下:

```
claritys-MacBook-Pro:first-network clarity$ tree channel-artifacts
channel-artifacts
├── Org1MSPanchors.tx
├── Org2MSPanchors.tx
├── channel.tx
└── genesis.block
```

0 directories, 4 files

其中几个参数说明如下。

- `genesis.block`: 排序服务创世区块;
- `channel.tx`: 通道配置创世区块;
- `Org1MSPanchors.tx`: Org1 锚节点的配置;
- `Org2MSPanchors.tx`: Org2 锚节点的配置。

### 11.2.4 启动超级账本网络

执行脚本 `./byfn.sh -m up` 会根据 Docker Compose 配置文件 `docker-compose-cli.yaml` 启动超级账本网络，还会执行 `scripts/script.sh` 脚本安装和实例化链码，并且执行简单链码调用和查询操作。

```
./byfn.sh -m up
```

输出的结果如下：

```
localhost:first-network clarity$ ./byfn.sh -m up
Starting with channel 'mychannel' and CLI timeout of '10000'
Continue (y/n)? y
proceeding ...
/Users/clarity/Projects/bin/cryptogen

#####
#####      使用 cryptogen 工具生成密钥和证书      #####
#####
org1.example.com
org2.example.com

/Users/clarity/Projects/bin/configtxgen
#####
#####      生成排序服务启动的创世区块      #####
#####
2017-08-09 19:53:51.192 CST [common/configtx/tool] main -> INFO 001 Loading configuration
2017-08-09 19:53:51.225 CST [common/configtx/tool] doOutputBlock -> INFO 002 Generating
genesis block
2017-08-09 19:53:51.227 CST [common/configtx/tool] doOutputBlock -> INFO 003 Writing
genesis block

#####
#####      生成通道配置交易文件 'channel.tx'      #####
#####
2017-08-09 19:53:51.253 CST [common/configtx/tool] main -> INFO 001 Loading configuration
2017-08-09 19:53:51.256 CST [common/configtx/tool] doOutputChannelCreateTx -> INFO
002 Generating new channel configtx
2017-08-09 19:53:51.256 CST [common/configtx/tool] doOutputChannelCreateTx -> INFO
003 Writing new channel tx

#####
#####      生成组织 Org1MSP 的锚节点配置      #####
#####
2017-08-09 19:53:51.280 CST [common/configtx/tool] main -> INFO 001 Loading configuration
2017-08-09 19:53:51.284 CST [common/configtx/tool] doOutputAnchorPeersUpdate -> INFO
002 Generating anchor peer update
2017-08-09 19:53:51.285 CST [common/configtx/tool] doOutputAnchorPeersUpdate -> INFO
003 Writing anchor peer update
```

```
#####
#####      生成组织 Org2MSP 的锚节点配置      #####
#####
2017-08-09 19:53:51.303 CST [common/configtx/tool] main -> INFO 001 Loading configuration
2017-08-09 19:53:51.306 CST [common/configtx/tool] doOutputAnchorPeersUpdate -> INFO
002 Generating anchor peer update
2017-08-09 19:53:51.306 CST [common/configtx/tool] doOutputAnchorPeersUpdate -> INFO
003 Writing anchor peer update

Creating network "net_byfn" with the default driver
Creating orderer.example.com ...
Creating peer1.org2.example.com ...
Creating peer0.org2.example.com ...
Creating peer0.org1.example.com ...
Creating peer1.org1.example.com ...
Creating peer1.org1.example.com
Creating peer0.org2.example.com
Creating orderer.example.com
Creating peer1.org2.example.com
Creating peer0.org1.example.com ... done
Creating cli ...
Creating cli ... done
```

看到上面的信息，表明超级账本网络已经启动完毕。在启动 cli 容器的时候会自动执行 scripts/script.sh 脚本，包括创建通道、Peer 节点加入通道、更新通道锚节点信息、安装链码、实例化链码、链码调用和链码查询等，每一步操作都会显示对应的日志信息。下面是创建通道的结果：

```

  _   _   _   _   _
 /_   _/   _/   _/   _/
/_   _/   _/   _/   _/
/_   _/   _/   _/   _/

Build your first network (BYFN) end-to-end test
```

```
Channel name : mychannel
Creating channel...
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/
crypto/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt
CORE_PEER_TLS_KEY_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/server.key
CORE_PEER_LOCALMSPID=Org1MSP
CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
CORE_PEER_TLS_CERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/server.crt
CORE_PEER_TLS_ENABLED=true
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/
crypto/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp
CORE_PEER_ID=cli
```



```
CORE_LOGGING_LEVEL=DEBUG
CORE_PEER_ADDRESS=peer0.org1.example.com:7051
...
```

```
===== Channel "mychannel" is created successfully =====
```

4 个 Peer 节点依次加入通道:

Having all peers join the channel...

```
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/
crypto/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt
CORE_PEER_TLS_KEY_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/server.key
CORE_PEER_LOCALMSPID=Org1MSP
CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
CORE_PEER_TLS_CERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/server.crt
CORE_PEER_TLS_ENABLED=true
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/
crypto/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp
CORE_PEER_ID=cli
CORE_LOGGING_LEVEL=DEBUG
CORE_PEER_ADDRESS=peer0.org1.example.com:7051
...
```

```
===== PEER0 joined on the channel "mychannel" =====
```

```
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/
crypto/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt
CORE_PEER_TLS_KEY_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/server.key
CORE_PEER_LOCALMSPID=Org1MSP
CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
CORE_PEER_TLS_CERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/server.crt
CORE_PEER_TLS_ENABLED=true
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/
crypto/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp
CORE_PEER_ID=cli
CORE_LOGGING_LEVEL=DEBUG
CORE_PEER_ADDRESS=peer1.org1.example.com:7051
...
```

```
===== PEER1 joined on the channel "mychannel" =====
```

```
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/
crypto/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt
CORE_PEER_TLS_KEY_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/server.key
CORE_PEER_LOCALMSPID=Org2MSP
CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
CORE_PEER_TLS_CERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/server.crt
CORE_PEER_TLS_ENABLED=true
```

```

CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/
crypto/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp
CORE_PEER_ID=cli
CORE_LOGGING_LEVEL=DEBUG
CORE_PEER_ADDRESS=peer0.org2.example.com:7051
...
===== PEER2 joined on the channel "mychannel" =====

CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/
crypto/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt
CORE_PEER_TLS_KEY_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/server.key
CORE_PEER_LOCALMSPID=Org2MSP
CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
CORE_PEER_TLS_CERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/server.crt
CORE_PEER_TLS_ENABLED=true
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/
crypto/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp
CORE_PEER_ID=cli
CORE_LOGGING_LEVEL=DEBUG
CORE_PEER_ADDRESS=peer1.org2.example.com:7051
...
===== PEER3 joined on the channel "mychannel" =====

```

更新组织的锚节点:

Updating anchor peers for org1...

```

CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/
crypto/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt
CORE_PEER_TLS_KEY_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/server.key
CORE_PEER_LOCALMSPID=Org1MSP
CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
CORE_PEER_TLS_CERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/server.crt
CORE_PEER_TLS_ENABLED=true
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/
crypto/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp
CORE_PEER_ID=cli
CORE_LOGGING_LEVEL=DEBUG
CORE_PEER_ADDRESS=peer0.org1.example.com:7051
...
=== Anchor peers for org "Org1MSP" on "mychannel" is updated successfully ===

```

Updating anchor peers for org2...

```

CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/
crypto/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt
CORE_PEER_TLS_KEY_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/server.key
CORE_PEER_LOCALMSPID=Org2MSP

```

```

CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
CORE_PEER_TLS_CERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/server.crt
CORE_PEER_TLS_ENABLED=true
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/
crypto/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp
CORE_PEER_ID=cli
CORE_LOGGING_LEVEL=DEBUG
CORE_PEER_ADDRESS=peer0.org2.example.com:7051
...
=== Anchor peers for org "Org2MSP" on "mychannel" is updated successfully ===

```

在 Peer 节点 peer0.org1.example.com、peer0.org2.example.com 上安装链码：

```

Installing chaincode on org1/peer0...
...
===== Chaincode is installed on remote peer PEER0 =====

Install chaincode on org2/peer2...
...
===== Chaincode is installed on remote peer PEER2 =====

```

在 Peer 节点 peer0.org2.example.com 上实例化链码：

```

Instantiating chaincode on org2/peer2...
...
===== Chaincode Instantiation on PEER2 on channel 'mychannel' is successful =====

```

调用前在 Peer 节点 peer0.org1.example.com 上执行链码查询 a 的值：

```

Querying chaincode on org1/peer0...
...
Query Result: 100
...
===== Query on PEER0 on channel 'mychannel' is successful =====

```

在 Peer 节点 peer0.org1.example.com 上执行链码调用，从 a 转移 10 到 b：

```

Sending invoke transaction on org1/peer0...
...
2017-09-10 02:56:13.627 UTC [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 00a
Chaincode invoke successful. result: status:200
...
===== Invoke transaction on PEER0 on channel 'mychannel' is successful =====

```

在 Peer 节点 peer1.org2.example.com 上安装链码并查询 a 的值：

```
Installing chaincode on org2/peer3...
...
===== Chaincode is installed on remote peer PEER3 =====

Querying chaincode on org2/peer3...
===== Querying on PEER3 on channel 'mychannel'... =====
...
Query Result: 90
...
===== Query on PEER3 on channel 'mychannel' is successful =====
```

可以看到，Peer 节点 peer1.org2.example.com 并没有参与背书的过程，在转移调用完成以后再安装链码，也能查询到最新的结果。到此，脚本自动执行的过程就结束了，超级账本网络还在继续运行：

```
===== All GOOD, BYFN execution completed =====
```



### 11.2.5 关闭超级账本网络

测试完成以后，调用脚本关闭超级账本网络：

```
./byfn.sh -m down
```

显示如下结果：

```
Stopping with channel 'mychannel' and CLI timeout of '10000'
Continue (y/n)? y
proceeding ...
WARNING: The CHANNEL_NAME variable is not set. Defaulting to a blank string.
WARNING: The TIMEOUT variable is not set. Defaulting to a blank string.
Stopping cli ... done
Stopping peer0.org1.example.com ... done
...
Removing cli ... done
Removing peer0.org1.example.com ... done
...
Removing network net_byfn
d1866048b1d5
b091ac28a1db
2a840ddfb530
Untagged: dev-peer1.org2.example.com-mycc-1.0:latest
Deleted: sha256:fd58f0b0679c2509acb668400c7b041d7d3a266f454e56bb1b49ff4484d30185
...
```

以上日志显示已经关闭 Peer 节点、排序服务节点容器和链码容器，删除自动生成的链码镜像文件。执行 `docker ps -a` 可以看到容器已经全部停止，执行 `docker images` 可以看到已经没有了 `dev-peer1.org2.example.com-mycc-1.0` 等容器镜像。

## 11.3 逐步建立超级账本网络

下面我们手工逐步建立超级账本网络，理解了后面的操作步骤，就可以根据需求自行定制和部署超级账本网络了。

### 11.3.1 生成 MSP 证书

使用 `cryptogen` 工具生成证书。MSP 证书是超级账本网络实体的身份标识，实体在通信和交易时使用证书进行签名和验证。生成证书需要 `crypto-config.yaml` 配置文件，详细的文件解析参考附录 B 的内容。这个文件定义了组织结构，据此可以为组织和其内的成员生成数字证书和签名密钥：

```
localhost:first-network clarity$ cryptogen generate
--config=./crypto-config.yaml
org1.example.com
org2.example.com
```

生成的 MSP 目录结构请参考第 8 章的内容。

### 11.3.2 生成排序服务创世区块

添加环境变量指定 `configtx.yaml` 文件的位置，生成创世区块：

```
export FABRIC_CFG_PATH=$PWD
configtxgen -profile TwoOrgsOrdererGenesis
-outputBlock ./channel-artifacts/genesis.block
```

执行结果如下：

```
2017-08-09 17:29:18.732 CST [common/configtx/tool] main -> INFO 001 Loading configuration
2017-08-09 17:29:18.761 CST [common/configtx/tool] doOutputBlock -> INFO 002 Generating
genesis block
2017-08-09 17:29:18.763 CST [common/configtx/tool] doOutputBlock -> INFO 003 Writing
genesis block
```

执行成功以后会在 `channel-artifacts` 目录下生成创世区块 `genesis.block`，区块内容包含了联盟和组织信息，还包含通道的访问控制策略信息，可以通过 `configtxlator` 或者 `configtxgen` 工具查看详细内容。

### 11.3.3 生成通道配置创世区块

设置环境变量通道名称，同样利用 `configtxgen` 生成通道配置：

```
export CHANNEL_NAME=mychannel
configtxgen -profile TwoOrgsChannel -outputCreateChannelTx ./channel-artifacts/
channel.tx -channelID $CHANNEL_NAME
```

执行结果如下：

```
2017-08-09 17:36:09.651 CST [common/configtx/tool] main -> INFO 001 Loading configuration
2017-08-09 17:36:09.654 CST [common/configtx/tool] doOutputChannelCreateTx -> INFO
002 Generating new channel configtx
2017-08-09 17:36:09.654 CST [common/configtx/tool] doOutputChannelCreateTx -> INFO
003 Writing new channel tx
```

执行成功以后会在 channel-artifacts 目录下生成通道配置 channel.tx，可以通过 configtxlator 或者 configtxgen 工具查看详细内容。

### 11.3.4 定义组织锚节点

定义 Org1 和 Org2 的两个锚节点：

```
export CHANNEL_NAME=mychannel
// 组织 Org1 的锚节点
configtxgen -profile TwoOrgsChannel -outputAnchorPeersUpdate ./channel-artifacts/
Org1MSPanchors.tx -channelID $CHANNEL_NAME -asOrg Org1MSP
// 组织 Org2 的锚节点
configtxgen -profile TwoOrgsChannel -outputAnchorPeersUpdate ./channel-artifacts/
Org2MSPanchors.tx -channelID $CHANNEL_NAME -asOrg Org2MSP
```

执行结果如下：

```
2017-08-09 18:28:36.129 CST [common/configtx/tool] main -> INFO 001 Loading configuration
2017-08-09 18:28:36.132 CST [common/configtx/tool] doOutputAnchorPeersUpdate -> INFO
002 Generating anchor peer update
2017-08-09 18:28:36.132 CST [common/configtx/tool] doOutputAnchorPeersUpdate -> INFO
003 Writing anchor peer update
```

执行成功以后会在 channel-artifacts 目录下生成锚节点配置 Org1MSPanchors.tx 和 Org2MSPanchors.tx，可以通过 configtxlator 或者 configtxgen 工具查看详细内容。

### 11.3.5 启动超级账本网络

使用 docker-compose 启动超级账本网络，需要的配置文件是 docker-compose-cli.yaml，定义了 1 个排序服务节点、4 个 Peer 节点。还有一个命令行容器 cli，默认通过和 peer0.org1.example.com 通信实现与超级账本网络交互，进行链码部署等操作，切换环境变量也会和其他 Peer 节点进行通信。这里还会用到之前已经编译或者下载成功的 Hyperledger Fabric v1.0.0 镜像文件，以及上面刚刚生成的创世区块 genesis.block 用于排序服务节点的启动。

由于启动 cli 容器的时候默认会执行 ./scripts/script.sh 脚本，所以需要先屏蔽启动脚本，再启动我们的网络：

```
// 屏蔽 ./scripts/script.sh 启动脚本
sed -i '' 's/command/#command/' docker-compose-cli.yaml
// 设置通道名称
CHANNEL_NAME=$CHANNEL_NAME
// 启动网络
docker-compose -f docker-compose-cli.yaml up -d
```

执行成功以后调用 `docker ps -a` 查看启动的容器如下（简化的输出信息）：

```
localhost:~$ docker ps -a
CONTAINER ID        IMAGE                                     NAMES
d294b6b493f9        hyperledger/fabric-tools               cli
86e686846262        hyperledger/fabric-peer               peer1.org2.example.com
7d1040f33537        hyperledger/fabric-peer               peer0.org2.example.com
78c4daafb95f        hyperledger/fabric-peer               peer1.org1.example.com
f902601d3b89        hyperledger/fabric-peer               peer0.org1.example.com
08a04df7cf36        hyperledger/fabric-orderer            orderer.example.com
```

### 11.3.6 创建并加入通道

网络启动以后，需要先创建通道，进入 `cli` 容器进行操作：

```
// 进入 cli 容器
docker exec -it cli bash
// 设置通道名称的环境变量
export CHANNEL_NAME=mychannel
// 创建通道
peer channel create -o orderer.example.com:7050 -c $CHANNEL_NAME -f ./channel-
artifacts/channel.tx --tls $CORE_PEER_TLS_ENABLED --cafile /opt/gopath/src/github.
com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/
orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem
```

执行成功会在 `cli` 容器的当前路径下生成以通道名称命名的 `mychannel.block`，这个是通道配置的创世区块，里面包含通道配置信息，加入通道操作时需要使用。

#### 1. Peer 节点 peer0.org1.example.com 加入通道

现在把 Peer 节点 `peer0.org1.example.com` 加入通道：

```
peer channel join -b mychannel.block
```

执行结果中包含“Peer joined the channel”，可见加入通道成功：

```
2017-08-09 13:07:16.816 UTC [msp] GetLocalMSP -> DEBU 001 Returning existing local MSP
2017-08-09 13:07:16.816 UTC [msp] GetDefaultSigningIdentity -> DEBU 002 Obtaining
default signing identity
2017-08-09 13:07:16.824 UTC [channelCmd] InitCmdFactory -> INFO 003 Endorser and
orderer connections initialized
2017-08-09 13:07:16.824 UTC [msp/identity] Sign -> DEBU 004 Sign: plaintext: 0A8
A070A5C08011A0C0884C996D00510...8DBFBB0981111A080A000A000A000A00
```



```

2017-08-09 13:07:16.824 UTC [msp/identity] Sign -> DEBU 005 Sign: digest: E10EC5
D95EE07F8E8793EF652E886F6398F30D9C510BB4420C6AEC0DE31612AD
2017-08-09 13:07:16.859 UTC [channelCmd] executeJoin -> INFO 006 Peer joined the
channel!
2017-08-09 13:07:16.859 UTC [main] main -> INFO 007 Exiting.....

```

查询当前 Peer 节点加入的通道列表:

```
peer channel list
```

日志中 "Channels peers has joined to" 后面显示的 mychannel 就是加入的通道:

```

2017-08-09 13:14:12.230 UTC [msp] GetLocalMSP -> DEBU 001 Returning existing local MSP
2017-08-09 13:14:12.230 UTC [msp] GetDefaultSigningIdentity -> DEBU 002 Obtaining
default signing identity
2017-08-09 13:14:12.237 UTC [channelCmd] InitCmdFactory -> INFO 003 Endorser and
orderer connections initialized
2017-08-09 13:14:12.237 UTC [msp/identity] Sign -> DEBU 004 Sign: plaintext: 0A8
9070A5B08031A0B08A4CC96D00510...631A0D0A0B4765744368616E6E656C73
2017-08-09 13:14:12.238 UTC [msp/identity] Sign -> DEBU 005 Sign: digest: 7E0B57
DC621486091E6DB68F41045957A83245292326472FACC493D5B077C3DE
2017-08-09 13:14:12.241 UTC [channelCmd] list -> INFO 006 Channels peers has
joined to:
2017-08-09 13:14:12.241 UTC [channelCmd] list -> INFO 007 mychannel
2017-08-09 13:14:12.241 UTC [main] main -> INFO 008 Exiting.....

```

## 2. Peer 节点 peer1.org1.example.com 加入通道

需要依次把其他 Peer 节点也加入通道, 在 cli 容器中切换环境变量, 设置连接的 Peer 节点和 MSP 等信息, 再加入通道:

```

// 设置连接的 Peer 节点
export CORE_PEER_ADDRESS=peer1.org1.example.com:7051
// 设置 MSP 信息
export CORE_PEER_LOCALMSPID="Org1MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/
peer/crypto/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt
export CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/
peer/crypto/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp
// 把 peer1.org1.example.com 加入通道
peer channel join -b mychannel.block

```

可以通过 peer channel list 查询到 Peer 节点 peer1.org1.example.com 已经成功加入通道 mychannel。

## 3. Peer 节点 peer0.org2.example.com 加入通道

需要先设置连接的 Peer 节点和 MSP 等信息, 再加入通道:

```

// 设置连接的 Peer 节点
export CORE_PEER_ADDRESS=peer0.org2.example.com:7051

```

```
// 设置 MSP 信息
export CORE_PEER_LOCALMSPID="Org2MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/
peer/crypto/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt
export CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/
peer/crypto/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp
// 把 peer1.org1.example.com 加入通道
peer channel join -b mychannel.block
```

可以通过 `peer channel list` 查询到 Peer 节点 `peer0.org2.example.com` 已经成功加入通道 `mychannel`。

#### 4. Peer 节点 `peer1.org2.example.com` 加入通道

需要先设置连接的 Peer 节点和 MSP 等信息，再加入通道：

```
// 设置连接的 Peer 节点
export CORE_PEER_ADDRESS=peer1.org2.example.com:7051
// 设置 MSP 信息
export CORE_PEER_LOCALMSPID="Org2MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/
peer/crypto/peerOrganizations/org2.example.com/peers/peer1.org2.example.com/tls/ca.crt
export CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/
peer/crypto/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp
// 把 peer1.org1.example.com 加入通道
peer channel join -b mychannel.block
```

可以通过 `peer channel list` 查询到 Peer 节点 `peer1.org2.example.com` 已经成功加入通道 `mychannel`。

### 11.3.7 安装和实例化链码

应用程序通过链码执行智能合约的功能，需要先在每个 Peer 节点上安装链码，然后在通道上实例化链码。

#### 1. Peer 节点 `peer0.org1.example.com` 安装链码

在 Peer 节点 `peer0.org1.example.com` 上安装链码，先设置环境变量，再安装链码 `chaincode_example02`：

```
// 设置连接的 Peer 节点
export CORE_PEER_ADDRESS=peer0.org1.example.com:7051
// 设置 MSP 信息
export CORE_PEER_LOCALMSPID="Org1MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/
peer/crypto/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt
export CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/
peer/crypto/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp
// 安装链码
peer chaincode install -n mycc -v 1.0 -p github.com/hyperledger/fabric/examples/
```

```
chaincode/go/chaincode_example02
```

执行结果中包含 “Installed remotely response” 说明执行成功：

```
2017-08-09 13:53:41.941 UTC [golang-platform] getCodeFromFS -> DEBU 005
getCodeFromFS
github.com/hyperledger/fabric/examples/chaincode/go/chaincode_example02
...
2017-08-09 13:53:42.123 UTC [chaincodeCmd] install -> DEBU 00d Installed remotely
response:<status:200 payload:"OK" >
2017-08-09 13:53:42.123 UTC [main] main -> INFO 00e Exiting.....
```

重新打开一个终端，进入 Peer 节点 peer0.org1.example.com 容器确认是否有安装的链码文件：

```
localhost:fabirc clarity$ docker exec peer0.org1.example.com ls /var/hyperledger/
production/chaincodes
```

如果输出结果中包含 mycc.1.0，就说明已经安装成功。

## 2. Peer 节点 peer1.org1.example.com 安装链码

在 Peer 节点 peer1.org1.example.com 上安装链码，先设置环境变量，再安装链码 chaincode\_example02：

```
// 设置连接的 Peer 节点
export CORE_PEER_ADDRESS=peer1.org1.example.com:7051
// 设置 MSP 信息
export CORE_PEER_LOCALMSPID="Org1MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/
peer/crypto/peerOrganizations/org1.example.com/peers/peer1.org1.example.com/tls/ca.crt
export CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/
peer/crypto/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp
// 安装链码
peer chaincode install -n mycc -v 1.0 -p github.com/hyperledger/fabric/examples/
chaincode/go/chaincode_example02
```

安装成功返回结果中会包含 “Installed remotely response:”，再进入 Peer 节点容器确认是否有安装的链码文件：

```
localhost:fabirc clarity$ docker exec peer1.org1.example.com ls /var/hyperledger/
production/chaincodes
```

如果输出结果中包含 mycc.1.0，就说明已经安装成功。

## 3. Peer 节点 peer0.org2.example.com 安装链码

在 Peer 节点 peer0.org2.example.com 上安装链码，先设置环境变量，再安装链码 chaincode\_example02：

```
// 设置连接的 Peer 节点
export CORE_PEER_ADDRESS=peer0.org2.example.com:7051
// 设置 MSP 信息
export CORE_PEER_LOCALMSPID="Org2MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/
peer/crypto/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/
ca.crt
export CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/
peer/crypto/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp
// 安装链码
peer chaincode install -n mycc -v 1.0 -p github.com/hyperledger/fabric/examples/
chaincode/go/chaincode_example02
```

安装成功返回结果中会包含 “Installed remotely response:”，再进入 Peer 节点容器确认是否有安装的链码文件：

```
localhost:fabric clarity$ docker exec peer0.org2.example.com ls /var/hyperledger/
production/chaincodes
```

如果输出结果中包含 mycc.1.0，就说明已经安装成功。

#### 4. Peer 节点 peer1.org2.example.com 安装链码

在 Peer 节点 peer1.org2.example.com 上安装链码，先设置环境变量，再安装链码 chaincode\_example02：

```
// 设置连接的 Peer 节点
export CORE_PEER_ADDRESS=peer1.org2.example.com:7051
// 设置 MSP 信息
export CORE_PEER_LOCALMSPID="Org2MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/
peer/crypto/peerOrganizations/org2.example.com/peers/peer1.org2.example.com/tls/
ca.crt
export CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/
crypto/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp
// 安装链码
peer chaincode install -n mycc -v 1.0 -p github.com/hyperledger/fabric/examples/
chaincode/go/chaincode_example02
```

安装成功返回结果中会包含 “Installed remotely response:”，再进入 Peer 节点容器确认是否有安装的链码文件：

```
localhost:fabric clarity$ docker exec peer1.org2.example.com ls /var/hyperledger/
production/chaincodes
```

如果输出结果中包含 mycc.1.0，就说明已经安装成功。

#### 5. 实例化链码

实例化链码只需要执行一次，任意一个 Peer 节点都可以处理实例化的请求，同样是在 cli 容器中执行如下操作：

```
peer chaincode instantiate -o orderer.example.com:7050 --tls $CORE_PEER_TLS_ENABLED --cafile /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C $CHANNEL_NAME -n mycc -v 1.0 -c '{"Args":["init","a","100","b","200"]}' -P "OR('Org1MSP.member','Org2MSP.member')"
```

其中, -c 参数确定了账户初始化情况为 a=100 和 b=200, -P 参数指定了背书策略, 链码的每个交易需要由 Org1MSP 或 Org2MSP 两个组织中的任意一个成员背书签名, 才能通过背书策略, 继续进入后续交易流程。

上面的命令执行完成以后没有错误提示就是成功的, 会启动链码的容器 dev-peer1.org2.example.com-mycc-1.0:

```
localhost:~$ docker ps -a
CONTAINER ID        IMAGE                                     NAMES
f851c41073b8       dev-peer1.org2.example.com-mycc-1.0    dev-peer1.org2.example.com-mycc-1.0
d294b6b493f9       hyperledger/fabric-tools              cli
86e686846262       hyperledger/fabric-peer              peer1.org2.example.com
7d1040f33537       hyperledger/fabric-peer              peer0.org2.example.com
78c4daafb95f       hyperledger/fabric-peer              peer1.org1.example.com
f902601d3b89       hyperledger/fabric-peer              peer0.org1.example.com
08a04df7cf36       hyperledger/fabric-orderer            orderer.example.com
```

### 11.3.8 执行链码查询

先查看当前的环境变量:

```
env | grep CORE_PEER_ADDRESS
```

输出结果是:

```
CORE_PEER_ADDRESS=peer1.org2.example.com:7051
```

表明 cli 容器目前是从 Peer 节点 peer1.org2.example.com 查询的:

```
peer chaincode query -C $CHANNEL_NAME -nmycc -c '{"Args":["query","a"]}'
peer chaincode query -C $CHANNEL_NAME -nmycc -c '{"Args":["query","b"]}'
```

可以看到 a 和 b 的账户分别返回了之前初始化的数值。

我们切换环境变量然后通过 peer0.org2.example.com 做同样的查询工作:

```
// 设置连接的 Peer 节点
export CORE_PEER_ADDRESS=peer0.org2.example.com:7051
// 设置 MSP 信息
export CORE_PEER_LOCALMSPID="Org2MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt
export CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/
```

```
crypto/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp
// 查询 a 和 b 的值
peer chaincode query -C $CHANNEL_NAME -nmycc -c '{"Args":["query","a"]}'
peer chaincode query -C $CHANNEL_NAME -nmycc -c '{"Args":["query","b"]}'
```

结果和从 Peer 节点 peer1.org2.example.com 查询是一样的, 由于 Peer 节点 peer0.org2.example.com 的链码之前没有调用过, 所以在做查询操作前会自行启动 dev-peer0.org2.example.com-mycc-1.0 的链码容器, 然后再完成查询:

```
localhost:first-network clarity$ docker ps -a
CONTAINER ID        IMAGE                                     NAMES
f851c41073b8        dev-peer1.org2.example.com-mycc-1.0    dev-peer1.org2.example.com-
mycc-1.0
8537e573f7a5        dev-peer0.org2.example.com-mycc-1.0    dev-peer0.org2.example.com-
mycc-1.0
d294b6b493f9        hyperledger/fabric-tools               cli
86e686846262        hyperledger/fabric-peer               peer1.org2.example.com
7d1040f33537        hyperledger/fabric-peer               peer0.org2.example.com
78c4daafb95f        hyperledger/fabric-peer               peer1.org1.example.com
f902601d3b89        hyperledger/fabric-peer               peer0.org1.example.com
08a04df7cf36        hyperledger/fabric-orderer            orderer.example.com
```

### 11.3.9 执行链码调用

链码调用也可以在任意一个节点上执行, 下面的操作完成从 a 账户转账 10 到 b 账户:

```
peer chaincode invoke -oorderer.example.com:7050 --tls $CORE_PEER_TLS_ENABLED
--cafile /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlsacerts/tlsca.
example.com-cert.pem -C $CHANNEL_NAME -n mycc -c '{"Args":["invoke","a","b","10"]}'
```

执行成功的结果包含 “Chaincode invoke successful. result: status:200”:

```
2017-08-09 15:25:32.202 UTC [msp] GetLocalMSP -> DEBU 001 Returning existing local
MSP
2017-08-09 15:25:32.202 UTC [msp] GetDefaultSigningIdentity -> DEBU 002 Obtaining
default signing identity
2017-08-09 15:25:32.208 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 003
Using default escc
2017-08-09 15:25:32.208 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 004
Using default vscc
2017-08-09 15:25:32.209 UTC [msp/identity] Sign -> DEBU 005 Sign: plaintext: 0A9
4070A6608031A0B08EC8997D00510...696E766F6B650A01610A01620A023130
2017-08-09 15:25:32.209 UTC [msp/identity] Sign -> DEBU 006 Sign: digest: E5807B
923EAB14944DFCE2268413765F629D489159E345F919C8171E1704C3BA
2017-08-09 15:25:32.224 UTC [msp/identity] Sign -> DEBU 007 Sign: plaintext: 0A9
4070A6608031A0B08EC8997D00510...0BF1F0306F009754B827D0419FF8B29F
2017-08-09 15:25:32.224 UTC [msp/identity] Sign -> DEBU 008 Sign: digest: 5C8256
B24D92447294F7AA0012412541712E2BC31E8C7F80AA2AEB0B670EB3CC
```



```

2017-08-09 15:25:32.229 UTC [chaincodeCmd] chaincodeInvokeOrQuery -> DEBU 009
ESCC invoke result: version:1 response:<status:200 message:"OK" >
payload:"\n \250.\245\310\004\ns\306p\334/\331\306x\2244\347\020\333|\352
\034\323\267M\2719\265\306\n\304q\022Y\nE\022\024\n\004lsc\022\014\n\n\
n\004mycc\022\002\010\001\022-\n\004mycc\022*\n\007\n\001a\022\002\010\001\n\
007\n\001b\022\002\010\001\032\007\n\001a\032\00290\032\010\n\001b\032\003210
\032\003\010\310\001\"013\022\004mycc\032\0031.0"
endorsement:<endorser:"\n\007Org2MSP\022\374\005-----BEGIN
-----\nMIICGCCAb+gAwIBAgIQBhvnD4KTVCKvkyVz4YIMqDAKBggqhkJOPQQDAjBzMQsw\nnCQYDVQQGEWJVUzETMBEGA1UECBMKQ2FsaWZvcn5pYTEwMBQGA1UEBxMNU2FuIEZy\nnYW5jaXNjbzEzMBCGA1UEChMQb3JnMi5leGFtcGxlLnNvbTEcMBoGA1UEAxMTY2Eu\nnb3JnMi5leGFtcGxlLnNvbTAeFw0xNzE
xMTAwOTExMDIaFw0yNzExMDgwOTExMDIa\nnMFSxCzAJBgNVBAYTAlVTMRMwEQYDVQQIEWpDYWxpZm9
ybmlhMRYYwFAYDVQQLHEw1LT\nnYW4gRnJhbmNpc2NvMR8wHQYDVQQLDEXZwZWVYMC5vcncyLmV4YW1wbGU
uY29tMFkw\nnEwYHKOZIZj0CAQYIKoZIZj0DAQcDQGAEEAnjBLY2Axe5FqwfB1lIKT3KSfYRCDJUm\nn7
IUWS0qOzmjwjbWhAh5+M3rGdE90lIyLgG1VY5T6Rgm5GdAladW51q6NNMeswDgYD\nnVR0PAQH/
BAQDAgeAMAwGA1UdEWEb/wQCAAAAwKwYDVR0jBCQwIoAgjdtfuR1T/7Bh\nnYIVSk87JB7i9hBMVxs5xr
TqpmGI+iwwCgYIKoZIZj0EAWIDRwAwRAIgUWz5CSZc\nnUjuOf9uzFx5Y2uCQoOzi23RPIyrSD5MgPWoC
IHpD6f5QVVYaU0yC1AadOiA81Mjff\nn+yb/gNSK2jelSiIC\nn-----END -----\n"
signature:"0D\002
h\212\000\034-\301\223\230\265\273\357\255\226\344\206\301\210\302T\341\257
\316<\310'\031U\177<B\002
<\010a\273\025[<\232r\254\275\225\234\002\024\217\013\361\3600o\000\227T\270'\32
0A\237\370\262\237" >
2017-08-09 15:25:32.230 UTC [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 00a
Chaincode invoke successful. result: status:200
2017-08-09 15:25:32.230 UTC [main] main -> INFO 00b Exiting.....

```

同样可以在任意一个 Peer 节点上查询转账后的结果。

## 11.4 本章小结

本章详细介绍了搭建超级账本网络的过程，内容由浅入深循序渐进，初学者能够快速体验超级账本的基本功能。再结合前面章节对内部实现原理的介绍，读者能够在详细的步骤中了解每一个步骤的作用，逐步深入地了解超级账本的内在魅力。



## Chapter 12 第12章

## 超级账本的应用开发实例

前面章节介绍了如何利用 Hyperledger Fabric Client SDK 和链码开发基于 Fabric 网络的区块链应用。本章会从应用开发的角度出发，用一个票据背书的例子来介绍如何进行一个完整的区块链应用开发。

## 12.1 票据背书场景介绍

这里仅讨论狭义的票据，指出票人依法签发的由自己或指示他人无条件支付一定金额给收款人或持票人的有价证券，分为汇票、本票和支票等。它们的共同特点是，在票据规定的期限内，持票人或收款人可向出票人或指定付款人无条件地支取确定金额的货币；它们都属于反映一定债权债务关系的、可流通的、代表一定数量货币请求权的有价证券。票据是在货币或商品流动中为体现债权和债务的发生、转移以及偿付而用的一种信用工具，可用作贸易中的支付结算和企业短期融资。票据的特征和意义如下。

1) 票据业务支持实体经济发展：票据承兑环节能为实体的企业支付、结算提供便利，满足企业间短期资金支付的需要。以企业间的背书转让和票据贴现为代表的交易方式能为实体经济（特别是中小企业）提供便捷的融资渠道和低成本资金。

2) 票据业务有助于推动货币市场发展：票据为金融机构之间的转贴现业务加快了短期资金的融通和调剂，已经成为银行等金融机构一项重要的资产业务。同时，以央行的再贴现、回购为代表的货币政策工具，使得票据在货币政策传导、增强货币政策实施效果、促进信贷机构调整、引导扩大中小企业融资范围等方面发挥了重要的作用。

3) 票据业务有助于丰富金融市场产品：由于票据兼有的支付、资金、信贷、资产等多

重属性，加上持票机构的多样化和跨领域流通，使得以票据为载体的衍生产品成为连接货币市场和资本市场的重要探索。

### 12.1.1 票据关系人

票据的主要关系人如表 12-1 所示。

表 12-1 票据主要的关系人

关系人名称	关系人说明
出票人	出票人是开立票据并将其交付给他人的法人、其他组织或者个人。出票人对收款人及正当持票人承担票据在提示付款或承兑时必须付款或者承兑的保证责任
付款人	付款人又称受票人。即指接受票据命令支付票款的人。他是票据的当事人。付款人对票据承担付款责任。付款人对票据承兑后便成为承兑人，即为票据的主债务人，他不能以任何借口解除其付款责任
收款人	收款人指收取票款的人。收款人有权要求出票人或付款人付款或承兑
承兑人	承兑人指在商业汇票上承诺并记载汇票上在汇票到期日支付汇票金额的付款人，也是汇票的主债务人
参加承兑人	参加承兑人是在票据提示后，遭到拒绝承兑或无法获得承兑时，非票据债务人作成参加承兑行为的人。票据到期时，如付款人拒不付款，参加承兑人负责支付票款
背书人	背书是一种票据行为，是票据转让的一种重要方式。背书是由持票人在汇票背面签上自己的名字，并将汇票交付给受让人的行为。这里的持票人称为背书人，受让人称为被背书人
持票人	持票人是指持有票据可依法向票据义务人主张票据权利（即要求对方付款）的人，包括拥有票据的收款人和从转让人手中取得票据的受让人
善意持票人	善意持票人是善意地付了全部金额的对价，取得一张表面完整、合格的、不过期的票据的持票人，他未发现这张票据曾被退票，也未曾发现其前手在权利方面有任何缺陷
保证人	保证人是以自己的名义对票据付款加以保证的人。保证人可以为出票人、背书人、承兑人或参加承兑人提供担保

票据在整个生命周期中并不一定能涉及上表中所有的关系人，最重要的几个关系人是出票人、付款人、收款人、承兑人、背书人等。

### 12.1.2 票据行为分类

票据行为有广义和狭义两种。广义的票据行为是指以发生、变更或消灭票据的权利义务关系为目的法律行为，包括出票、背书、涂改、禁止背书、付款、保证、承兑、参加承兑、划线、保付等。狭义的票据行为是票据当事人以负担票据债务为目的的法律行为，包括出票、背书、承兑、参加承兑、保证、保付等。

1) 出票。出票是指出票人依照法定款式做成票据并交付于受款人的行为。它包括“做成”和“交付”两种行为。所谓“做成”就是出票人按照法定款式制作票据，在票据上记载法定内容并签名。由于现在各种票据都由一定机关印制，因此所谓“做成”只是填写有关内容和签名而已。所谓“交付”是指根据出票人本人的意愿将其交给受款人的行为，不是出于出票人本人意愿的行为（如偷窃票据）不能称作“交付”，因此也不能称作出票行为。

2) 背书。背书是指持票人转让票据权利与他人。票据的特点在于其流通。票据转让的主要方法是背书,当然,除此之外,还有单纯交付。背书转让是持票人的票据行为,只有持票人才能进行票据的背书。背书是转让票据权利的行为,票据一经背书转让,票据上的权利也随之转让给被背书人。

3) 承兑。承兑是指汇票的付款人承诺负担票据债务的行为。承兑为汇票所独有。汇票的发票人和付款人之间是一种委托关系,发票人签发汇票,并不等于付款人就一定付款,持票人为确定汇票到期时能得到付款,在汇票到期前向付款人进行承兑提示。如果付款人签字承兑,那么他就对汇票的到期付款承担责任,否则持票人有权对其提起诉讼。

4) 参加承兑。参加承兑是指票据的预备付款人或第三人为了特定票据债务人的利益,代替承兑人进行承兑,以阻止持票人于汇票到期日前行使追索权的一种票据行为。它一般是在汇票得不到承兑、付款人或承兑人死亡、逃亡或其他原因无法承兑、付款人或承兑人被宣告破产的情况下发生。

6) 保证。保证是指除票据债务人以外的人为担保票据债务的履行、以负担同一内容的票据债务为目的的一种附属票据行为。票据保证的目的是担保其他票据债务的履行,适用于汇票和本票,不适用于支票。

7) 保付。保付是指支票的付款人向持票人承诺负绝对付款责任的一种附属票据行为。保付是支票付款人的一种票据行为。支票一旦经付款人保付,在支票上注明“照付”或“保付”字样,并经签名后,付款人便负绝对付款责任,不论发票人在付款人处是否有资金,也不论持票人在法定提示期间是否有提示,或者即使发票人撤回付款委托,付款人均须按规定付款。

8) 贴现。贴现指银行承兑汇票的持票人在汇票到期日前,为了取得资金,贴付一定利息将票据权利转让给银行的票据行为,是持票人向银行融通资金的一种方式。

9) 转贴现。转贴现指商业银行在资金临时不足时,将已经贴现但仍未到期的票据,交给其他商业银行或贴现机构给予贴现,以取得资金融通。

10) 再贴现。再贴现指中央银行通过买进商业银行持有的已贴现但尚未到期的商业汇票,向商业银行提供融资支持的行为。

在具体操作时,票据行为表现为票据当事人把行为的意思按照法定的方式记载在票据上,并由行为人签章后将票据交付。它包括三方面内容,即记载、签章和交付。

1) 记载,通俗地讲,就是票据当事人在票据上写明所要记载的内容,如签发票据时应写明票据的种类、金额、无条件支付命令、签发票据日期以及其他需要明确的内容,承兑汇票时写上“承兑”字样,保证时应写上“保证”或“担保”字样。

2) 签章,指签名、盖章或签名加盖章,它表明行为人对其行为承担责任。自然人签章是指在票据上亲自书写其姓名或加盖其私章。法人和其他使用票据单位的签章为该法人或者该单位的盖章加其法定代表人或其授权的代理人的签章。按照《票据法》规定,在票据上的签名应当为该当事人的本名,而不能用笔名、艺名等来代替。

3) 交付, 是指票据行为人应将票据交付给执票人。票据行为人在票据上进行记载, 并进行签章后, 票据还不能发生法律效力, 只有把票据交付给了对方, 票据才能发生法律效力。

### 12.1.3 基于区块链技术的数字票据

票据现有的形式有纸质票据和电子票据。纸质票据是传统的票据形式, 需要在票据上签字或者加盖有效印章才能生效。电子票据是基于央行牵头开发完成的电子商业汇票系统 (ECDS), 银行或者企业通过直连或者网银接入, 所有的票据承兑、交易等都需要通过 ECDS 才能完成, 是典型的中心化系统。纸质票据和电子票据都有一些尚未解决的痛点。

- 票据真实性问题: 票据的贸易背景信息可能并不真实存在或者存在偏差, 票据信息也容易被克隆和伪造, 市场中存在的假票很难识别。
- 票据安全性问题: 纸质票据在携带过程有遗失或者损坏的风险, 也存在打款和背书不同步的情况。
- 票据信用风险问题: 可能存在汇票到期后承兑人未及时将相关款项划入持票人账户的情况。
- 票据处理效率问题: 在途时间会造成资金结算延后, 监管和审计成本也会很高。
- 票据违规交易问题: 票据交易主体或者中介机构可能存在一票多卖、清单交易、出租账户等违法违规行为。

基于区块链技术的数字票据能较好地解决以上问题, 如图 12-1 所示。

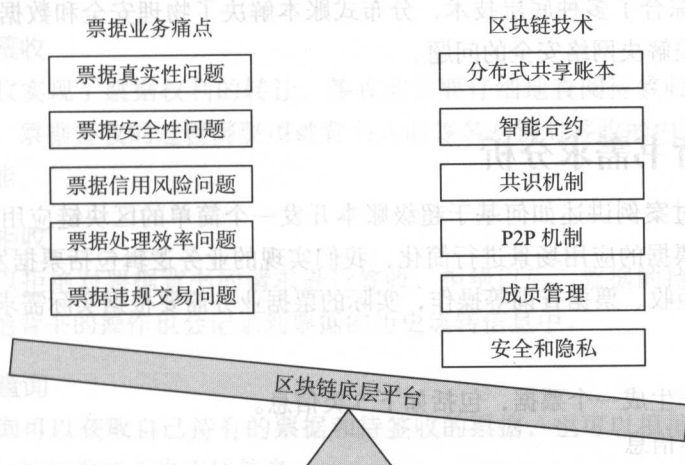


图 12-1 基于区块链技术的数字票据业务

借助区块链技术, 可以在实现原有的票据业务的基础上解决现有的一些痛点。

(1) 在不依赖可信第三方平台的基础上实现票据信息的真实性

纸质票据和电子票据的交易双方都依赖可信的第三方平台或者票据实物验证票据的真

伪。但是在实际的操作过程中,票据的真实信息是很难确认的。超级账本的成员管理利用底层的安全机制确保上链信息的真实性,基于记录到分布式共享账本中的票据信息,杜绝假票、克隆票、变造票等伪假票据。

### (2) 分布式共享账本避免违规交易

纸质票据中的一票多卖和电子票据中打款背书不同步等违规交易存在的原因是信息的不同步。基于区块链技术的数字票据利用分布式共享账本记录的信息具有不可篡改性,一旦交易完成并记录到账本中,就不会存在纸质票据和电子票据中的数据丢失和信息不同步的问题,交易双方都没有机会进行违规交易。

### (3) 智能合约实现票据业务自动处理

智能合约功能可以实现复杂的业务逻辑,比如业务操作前可以基于不可篡改的账本信息搜集和评估参与者的信用,拒绝信用评级低的交易方提交的请求,业务操作后可以自动实现转账等功能,解决不同操作导致的信息不同步问题。

### (4) 数字票据提升票据处理效率

分布式共享账本在区块链网络中有多份数据拷贝,交易结束后就完成了对账,利用本地的账本能快速检索信息,做更多的业务处理,提升票据处理的效率。

### (5) 区块链技术提高票据安全性

票据安全性包括物理安全、数据安全、网络安全等多个方面。纸质票据在携带过程有遗失或者损坏的风险,属于物理安全范畴。纸质票据和电子票据都可能存在数据被篡改的可能性,属于数据安全范畴。传输过程中被中间人攻击等属于网络安全范畴。基于区块链技术的数字票据综合了多种底层技术,分布式账本解决了物理安全 and 数据安全的问题,数字签名和安全传输解决网络安全的问题。

## 12.2 票据背书需求分析

本章旨在通过案例讲述如何基于超级账本开发一个简单的区块链应用,票据背书的应用开发实例会对票据的应用场景进行简化,我们实现的业务逻辑包括票据发布、票据背书、票据签收、票据拒收、票据查询等操作,实际的票据业务需要根据实际需求做调整。

### 1. 票据发布

票据发布操作生成一个票据,包括如下5类信息。

#### (1) 票据基本信息

☐ 票据号码

☐ 票据金额

☐ 票据种类

☐ 票据出票日期

☐ 票据到期日期



### (2) 出票人信息

- ☐ 出票人名称
- ☐ 出票人证件号码

### (3) 承兑人信息

- ☐ 承兑人名称
- ☐ 承兑人证件号码

### (4) 收款人信息

- ☐ 收款人名称
- ☐ 收款人证件号码

### (5) 持票人信息

- ☐ 持票人名称
- ☐ 持票人证件号码

## 2. 票据背书

票据背书是转让票据权利的重要方式和手段。发票票据需要先获取持票人持有的票据，填写被背书人的信息：

- ☐ 被背书人名称；
- ☐ 被背书人的证件号码。

发起票据背书的请求后会提交给被背书人。被背书人接收到票据背书的请求后，可以选择签收票据或者拒绝背书。

## 3. 票据背书签收

票据背书签收实现了票据权利的转让，签收前需要仔细地查阅待签收票据的信息，确保内容的完整性。票据签收的过程需要用被背书人的签名密钥对签收的内容进行数字签名，实现防抵赖的功能。

## 4. 票据背书拒收

被背书人可以拒绝对票据背书的请求进行签收，拒绝以后，票据的持有人还是票据背书的发起者。拒绝背书的操作也会记录到票据的历史流转信息中。

## 5. 票据信息查询

票据信息查询可以获取自己持有的票据和待签收的票据，也可以根据票据号码查询票据的详细信息，包括票据的历史流转信息。

# 12.3 票据背书架构设计

根据票据背书的需求分析，本节设计一个简单的架构，再定义票据背书的数据模型。

12.3.1 票据背书的分层架构

我们利用图 10.1 所示 Hyperledger Fabric 1.0 的应用开发模型来实现票据背书的应用场景。我们将基于区块链的数字票据进行分层设计，包括 Hyperledger Fabric 1.0 底层平台、智能合约、业务层和应用层，如图 12-2 所示。

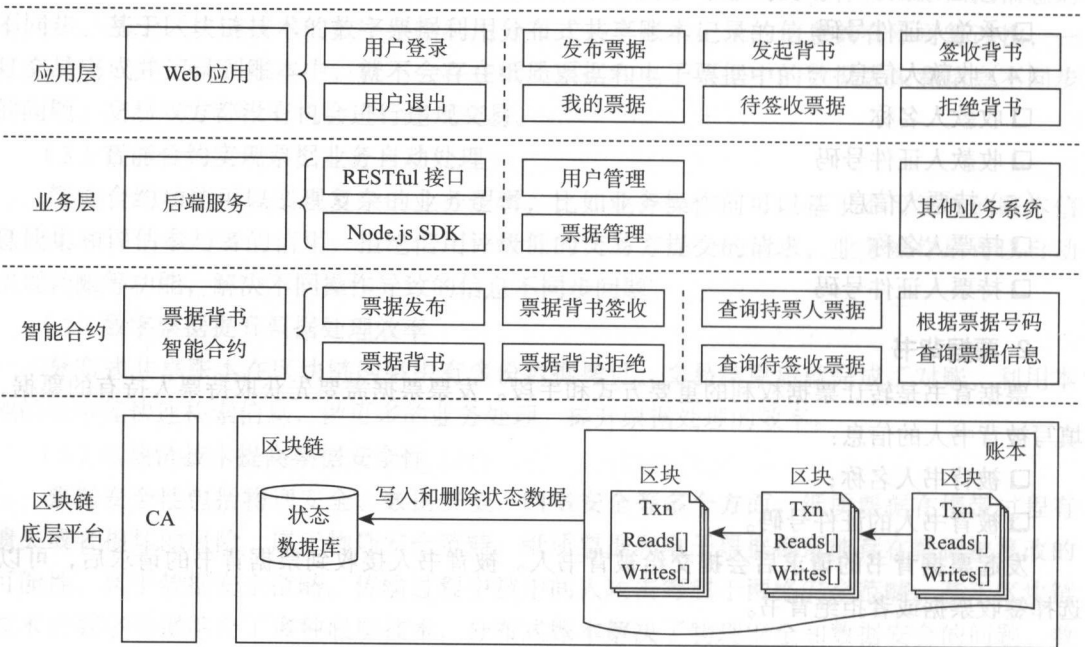


图 12-2 票据背书的分层架构

每个层的主要功能如下。

- 1) **区块链底层平台**：提供分布式共享账本的维护、状态数据库维护、智能合约的全生命周期管理等区块链功能，实现数据的不可篡改和智能合约的业务逻辑。根据第 11 章的内容搭建区块链网络以后，默认就提供了这部分功能。另外，通过 fabric-ca 提供成员注册和注销等功能。
- 2) **智能合约**：智能合约通过链码来实现，包括票据发布、票据背书、票据背书签收、票据背书拒绝等链码调用功能，链码查询包括查询持票人票据、查询待签收票据、根据链码号码查询票据信息等。票据系统的其他功能（比如贴现、转贴现、再贴现、回购等一系列业务类型）都可以在智能合约里实现，这部分功能留给读者继续完善。
- 3) **业务层**：业务层是应用程序的后端服务，给 Web 应用提供 RESTful 的接口，处理前端的业务请求。后端服务的基本功能包括用户管理和票据管理，通过 Hyperledger Fabric 1.0 提供的 Node.js SDK 和区块链网络进行通信。业务层也可以和其他的业务系统进行交互。



4) 应用层: Web 应用采用 Angular.js+HTML+CSS 的前端架构编写具有 MVC、模块化、自动数据绑定等特点的单页面应用, 提供用户交互的界面操作, 包括用户操作的功能和业务操作的功能。用户是内置的, 只提供用户登录和用户退出操作。业务操作包括发布票据、查询持票人持有的票据、发起票据背书、查询待签收票据、签收票据背书、拒绝票据背书等功能。

各个层之间采用不同的接口, 业务层的 Node.js SDK、智能合约和区块链底层平台之间采用 gRPC 的接口, 业务层和 Web 应用之间采用 RESTful 的接口。

### 12.3.2 票据背书的数据模型

本节看一下链码设计的数据模型, 包括票据数据结构定义和票据状态定义。

#### 1. 票据数据结构

票据信息包括票据基本信息、出票人信息、承兑人信息、收款人信息、持票人信息、待背书人信息、拒绝背书人信息、票据状态和票据背书历史等, 数据结构定义如下:

```
// 票据数据结构
type Bill struct {
    BillInfoID string `json:BillInfoID`           // 票据号码
    BillInfoAmt string `json:BillInfoAmt`         // 票据金额
    BillInfoType string `json:BillInfoType`        // 票据类型
    BillInfoIsseDate string `json:BillInfoIsseDate`    // 票据出票日期
    BillInfoDueDate string `json:BillInfoDueDate`    // 票据到期日期
    DrwrCmID string `json:DrwrCmID`               // 出票人证件号码
    DrwrAcct string `json:DrwrAcct`                // 出票人名称
    AccepCmID string `json:AccepCmID`              // 承兑人证件号码
    AccepAcct string `json:AccepAcct`              // 承兑人名称
    PyeeCmID string `json:PyeeCmID`               // 收款人证件号码
    PyeeAcct string `json:PyeeAcct`               // 收款人名称
    HodrCmID string `json:HodrCmID`               // 持票人证件号码
    HodrAcct string `json:HodrAcct`               // 持票人名称
    WaitEndorserCmID string `json:WaitEndorserCmID` // 待背书人证件号码
    WaitEndorserAcct string `json:WaitEndorserAcct` // 待背书人名称
    RejectEndorserCmID string `json:RejectEndorserCmID` // 拒绝背书人证件号码
    RejectEndorserAcct string `json:RejectEndorserAcct` // 拒绝背书人名称
    State string `json:State`                     // 票据状态
    History []HistoryItem `json:History`          // 票据背书历史
}
```

票据历史信息包含了票据的流转信息, 比如票据发布、票据签收、票据拒绝等都会记录到历史信息中, 数据结构定义如下:

```
// 背书历史 item 结构
type HistoryItem struct {
    TxId string `json:"txId"`
    Bill Bill `json:"bill"`
}
```

票据历史信息是智能合约自动完成的。

2. 票据状态模型

票据状态定义如表 12-2 所示。

表 12-2 票据状态定义

票据状态	状态说明	票据状态	状态说明
NewPublish	票据新发布	EndrSigned	票据签收成功
EndrWaitSign	票据等待签收	EndrReject	票据被拒绝签收

票据背书的状态转移图如图 12-3 所示。

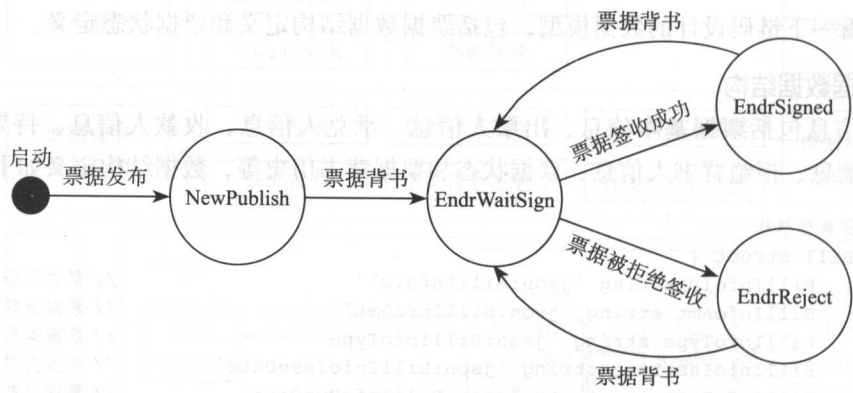


图 12-3 票据背书的状态转移图

票据发布以后进入票据新发布状态 NewPublish，票据持票人可以提交票据背书的操作进行票据权利转移，进入票据等待签收的状态 EndrWaitSign。票据被背书人接收到票据背书请求后，可以选择签收票据或者拒绝签收，票据签收成功进入状态 EndrSigned，持票人转移为被背书人；拒绝签收进入状态 EndrReject，持票人保持不变还是原有的持票人。处于 EndrSigned 和 EndrReject 状态的持票人都可以再次发起票据背书的请求，进入下一轮的操作。

12.4 票据背书实现

票据背书的实现分为两个部分，即基于 Hyperledger Fabric Node.js SDK 的应用程序和链码功能的实现。本章所有的代码托管到 Github 上：<https://github.com/ChainNova/trainingProjects/tree/master/billEndorse>。后面只介绍部分业务逻辑的实现。

12.4.1 应用程序实现

应用程序分为 Web 应用前端和后端服务。这里只介绍后端服务的实现，Web 应用前端



记成 000001。若区块链上已经有该票据，输出为错误，提示为“票据重复发布”。

票据操作接口的 URL 都是相同的：`http://ip:port/channels/mychannel/chaincodes/mycc/invoke`，其中，ip 和 port 是 Web 应用的地址，mychannel 是通道名称，mycc 是链码的名称，这些参数都需要根据实际的部署做修改。

票据操作调用的接口通过 Body 信息来区分：

```
{
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE1MTIxNDM0MzAsInVzZXJuYWllIjoieWxpyY2UiLCJvcmdOYWllIjoib3JnMSIsInBhc3N3b3JkIjoimTIzNDU2IiwiaWF0IjoxNTEyMTA3NDMwfwQ.QnIyaxuq8G4Jl0lBNq3DMKYfs6q8zjLUYwRjxS1GdxU",
  "peers": ["peer1"],
  "fcn": "issue",
  "args": [{"BillInfoID": "POC10000998", "BillInfoAmt": "222", "BillInfoType": "111", "BillInfoIsseDate": "20170910", "BillInfoDueDate": "20171112", "DrwrCmID": "111", "DrwrAcct": "111", "AccptrCmID": "111", "AccptrAcct": "111", "PyeeCmID": "111", "PyeeAcct": "111", "HodrCmID": "ACMID", "HodrAcct": "A 公司"}]
}
```

其中，票据操作接口是通用的结构，各参数的含义如表 12-3 所示。

表 12-3 后端服务的接口参数定义

参数名称	参数说明
token	接口调用的认证信息，调用用户登录接口返回的
peers	背书节点的名称列表，在配置文件 network-config.json 中定义的
fcn	链码调用的函数名称，票据发布是固定的 issue，其他接口以示例为准
args	链码调用的参数，每个接口内容不一样，详细的解释参考 12.4.2 节

返回信息如下：

```
{
  "success": true,
  "message": "9a1525ef5a388530c1757c9c1c565bf52422e9a775a03d20e9aa2273b008aa31"
}
```

(3) 票据背书接口

票据背书接口输入的 Body 信息如下：

```
{
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE1MTIxNDM0MzAsInVzZXJuYWllIjoieWxpyY2UiLCJvcmdOYWllIjoib3JnMSIsInBhc3N3b3JkIjoimTIzNDU2IiwiaWF0IjoxNTEyMTA3NDMwfwQ.QnIyaxuq8G4Jl0lBNq3DMKYfs6q8zjLUYwRjxS1GdxU",
  "peers": ["peer1"],
  "fcn": "endorse",
  "args": ["POC10000998", "BCMID", "B 公司"]
}
```

其中, 票据背书的 fcn 是 endorse, args 参数的信息参考 12.4.2 节。  
返回的信息如下:

```
{
  "success": true,
  "message": "ae87c2e1d51f22125e9c16375420aee68acd0bb3dbcb5950a787e4a5cba3b080"
}
```

#### (4) 票据背书签收接口

票据背书签收接口输入的 Body 信息如下:

```
{
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE1MTIxNDM4MDMsInVzZXJuYVllIjoiYm9iIiwib3JnTmFtZSI6Im9yZzEiLCJwYXNzd29yZCI6IjEyMzQ1NiIsImh0dCI6MTUxMjEwNzgwM30.RMxkdTOP2e6K03hD_6GpkHV3mcZpeqjcxfdqshb7gKk",
  "peers": ["peer1"],
  "fcn": "accept",
  "args": ["POC10000998", "BCMID", "B 公司"]
}
```

其中, 票据背书签收的用户 bob 需要先登录并获取 token, 票据背书签收的 fcn 是 accept, args 参数的信息参考 12.4.2 节。

返回的信息如下:

```
{
  "success": true,
  "message": "3710f07807521218f4ccadcd06c7ffba21f44fc2f70a773d3bc707fe48d7f99"
}
```

#### (5) 票据背书拒收接口

票据背书拒收接口输入的 Body 信息如下:

```
{
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE1MTIwODkwMjcsInVzZXJuYVllIjoiYWxpY2UiLCJvcmd0YXNpdCI6Im9yZzEiLCJwYXNzd29yZCI6IjEyMzQ1NiIsImh0dCI6MTUxMjEwNzgwM30.RMxkdTOP2e6K03hD_6GpkHV3mcZpeqjcxfdqshb7gKk",
  "peers": ["peer1"],
  "fcn": "reject",
  "args": ["POC10000998", "BCMID", "B 公司"]
}
```

其中, 票据背书拒收的 fcn 是 reject, args 参数的信息参考 12.4.2 节。

返回信息如下:

```
{
  "success": true,
  "message": "183b9ea86804f1fbf1cdd172210b612c89514e9266b082d54b5acda5be4b2f69"
}
```

## (6) 查询持票人的票据列表接口

查询持票人的票据列表接口输入的 Body 信息如下:

```
{
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE1MTIxNDM4MDMsI
nVzZXJuYW11IjoiYm9iIiwib3JnTmFtZSI6Im9yZzEiLCJwYXNzd29yZCI6IjEyMzQ1NiIsI
mlhdCI6MTUxMjEwNzgwM30.RMxkdTOP2e6K03hD_6GpkHV3mcZpeqjcxfdqshb7gKk",
  "peers": ["peer1"],
  "fcn": "queryMyBill",
  "args": ["BCMID"]
}
```

其中, 查询持票人的票据列表的 fcn 是 queryMyBill, args 参数的信息参考 12.4.2 节。

返回的信息如下:

```
{
  "success": true,
  "message": "[
    {
      'BillInfoID': 'POC10000998',
      'BillInfoAmt': '222',
      'BillInfoType': '111',
      'BillInfoIsseDate': '111',
      'BillInfoDueDate': '111',
      'DrwrCmID': '111',
      'DrwrAcct': '111',
      'AccptrCmID': '111',
      'AccptrAcct': '111',
      'PyeeCmID': '111',
      'PyeeAcct': '111',
      'HodrCmID': 'BCMID',
      'HodrAcct': 'B 公司',
      'WaitEndorserCmID': '',
      'WaitEndorserAcct': '',
      'RejectEndorserCmID': '',
      'RejectEndorserAcct': '',
      'State': 'EndrSigned',
      'History': null
    }
  ]"
```

说明一下, 为了方便阅读, 上面的返回信息对结果做了格式化处理, 把原始的结果中双引号的转义 “\” 替换成了单引号 “'”, 后面的展示结果也做了相同的处理。

## (7) 查询待签收票据列表接口

查询待签收票据列表接口输入的 Body 信息如下:

```
{
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE1MTIxNDM4MDMsI
```

```

nVzZXJuYVllIjoiYm9iIiwib3JnTmFtZSI6Im9yZzEiLCJwYXNzd29yZCI6IjEyMzQ1NiIsI
mlhdCI6MTUxMjEwNzgwM30.RMxkdTOP2e6K03hd_6GpkHV3mcZpeqjcxfdqshb7gKk",
"peers": ["peer1"],
"fcn": "queryMyWaitBill",
"args": ["BCMID"]
}

```

其中，查询待签收票据列表的 fcn 是 queryMyWaitBill，args 参数的信息参考 12.4.2 节。返回的信息如下：

```

{
  "success": true,
  "message": "[
    {
      'BillInfoID': 'POC10000999',
      'BillInfoAmt': '222',
      'BillInfoType': '111',
      'BillInfoIsseDate': '111',
      'BillInfoDueDate': '111',
      'DrwrCmID': '111',
      'DrwrAcct': '111',
      'AccptrCmID': '111',
      'AccptrAcct': '111',
      'PyeeCmID': '111',
      'PyeeAcct': '111',
      'HodrCmID': 'ACMID',
      'HodrAcct': 'A 公司',
      'WaitEndorserCmID': 'BCMID',
      'WaitEndorserAcct': 'B 公司',
      'RejectEndorserCmID': '',
      'RejectEndorserAcct': '',
      'State': 'EndrWaitSign',
      'History': null
    }
  ]"
}

```

#### (8) 根据票据号码查询票据信息接口

根据票据号码查询票据信息接口输入的 Body 信息如下：

```

{
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjEiLCJwYXNzd29yZCI6IjEyMzQ1NiIsI
nVzZXJuYVllIjoiYm9iIiwib3JnTmFtZSI6Im9yZzEiLCJwYXNzd29yZCI6IjEyMzQ1NiIsI
mlhdCI6MTUxMjEwNzgwM30.RMxkdTOP2e6K03hd_6GpkHV3mcZpeqjcxfdqshb7gKk",
  "peers": ["peer1"],
  "fcn": "queryByBillNo",
  "args": ["POC10000998"]
}

```

其中，根据票据号码查询票据信息的 fcn 是 queryByBillNo，args 参数的信息参考 12.4.2 节。



返回的信息如下:

```
{
  "success": true,
  "message": "{
    'BillInfoID': 'POC10000998',
    'BillInfoAmt': '222',
    'BillInfoType': '111',
    'BillInfoIsseDate': '111',
    'BillInfoDueDate': '111',
    'DrwrCmID': '111',
    'DrwrAcct': '111',
    'AccptrCmID': '111',
    'AccptrAcct': '111',
    'PyeeCmID': '111',
    'PyeeAcct': '111',
    'HodrCmID': 'BCMID',
    'HodrAcct': 'B 公司',
    'WaitEndorserCmID': '',
    'WaitEndorserAcct': '',
    'RejectEndorserCmID': '',
    'RejectEndorserAcct': '',
    'State': 'EndrSigned',
    'History': [
      {
        'txId': '9a1525ef5a388530c1757c9c1c565bf52422e9a775a03d20e9aa2273
b008aa31',
        'bill': {
          'BillInfoID': 'POC10000998',
          'BillInfoAmt': '222',
          'BillInfoType': '111',
          'BillInfoIsseDate': '111',
          'BillInfoDueDate': '111',
          'DrwrCmID': '111',
          'DrwrAcct': '111',
          'AccptrCmID': '111',
          'AccptrAcct': '111',
          'PyeeCmID': '111',
          'PyeeAcct': '111',
          'HodrCmID': 'ACMID',
          'HodrAcct': 'A 公司',
          'WaitEndorserCmID': '',
          'WaitEndorserAcct': '',
          'RejectEndorserCmID': '',
          'RejectEndorserAcct': '',
          'State': 'NewPublish',
          'History': null
        }
      },
      {
        'txId': 'ae87c2e1d51f22125e9c16375420aee68acd0bb3dbcb5950a787e4a5
```

```
cbasb080',  
    'bill': {  
        'BillInfoID': 'POC10000998',  
        'BillInfoAmt': '222',  
        'BillInfoType': '111',  
        'BillInfoIsseDate': '111',  
        'BillInfoDueDate': '111',  
        'DrwrCmID': '111',  
        'DrwrAcct': '111',  
        'AccptrCmID': '111',  
        'AccptrAcct': '111',  
        'PyeeCmID': '111',  
        'PyeeAcct': '111',  
        'HodrCmID': 'ACMID',  
        'HodrAcct': 'A 公司',  
        'WaitEndorserCmID': 'BCMID',  
        'WaitEndorserAcct': 'B 公司',  
        'RejectEndorserCmID': '',  
        'RejectEndorserAcct': '',  
        'State': 'EndrWaitSign',  
        'History': null  
    }  
},  
}  
  
'txId': '3710f07807521218f4ccadcd06c7ffba21f44fc2f70a773d3bc707fe48d7f99',  
    'bill': {  
        'BillInfoID': 'POC10000998',  
        'BillInfoAmt': '222',  
        'BillInfoType': '111',  
        'BillInfoIsseDate': '111',  
        'BillInfoDueDate': '111',  
        'DrwrCmID': '111',  
        'DrwrAcct': '111',  
        'AccptrCmID': '111',  
        'AccptrAcct': '111',  
        'PyeeCmID': '111',  
        'PyeeAcct': '111',  
        'HodrCmID': 'BCMID',  
        'HodrAcct': 'B 公司',  
        'WaitEndorserCmID': '',  
        'WaitEndorserAcct': '',  
        'RejectEndorserCmID': '',  
        'RejectEndorserAcct': '',  
        'State': 'EndrSigned',  
        'History': null  
    }  
}  
}"
```

## 2. HFC Node.js SDK 的使用

HFC Node.js SDK 的使用包括创建通道、加入通道、安装链码、实例化链码、调用链码等。

### (1) 创建通道

首先介绍 `getOrgAdmin` 函数，其目的是根据传入的 `orgName` 将 `client` 实例设置为对应该组织，针对后面的操作进行组织的配置，它作为 `util` 函数会在后面多次用到，具体实现内容如下：

- 1) 将 `client` 实例的 `CryptoSuite` 切换为传入组织的 `CryptoSuite`;
- 2) 将 `client` 实例的 `StateStore` 切换为传入组织的 `StateStore`;
- 3) 返回传入组织的 `admin` 用户实例。

```
var getOrgAdmin = function(userOrg) {
    var admin = ORGS[userOrg].admin;
    var keyPath = path.join(__dirname, admin.key);
    var keyPEM = Buffer.from(readAllFiles(keyPath)[0]).toString();
    var certPath = path.join(__dirname, admin.cert);
    var certPEM = readAllFiles(certPath)[0].toString();

    var client = getClientForOrg(userOrg);
    var cryptoSuite = hfc.newCryptoSuite();
    if (userOrg) {
        cryptoSuite.setCryptoKeyStore(hfc.newCryptoKeyStore({path: getKeyStoreForOrg(getOrgName(userOrg))}));
        client.setCryptoSuite(cryptoSuite);
    }

    return hfc.newDefaultKeyValueStore({
        path: getKeyStoreForOrg(getOrgName(userOrg))
    }).then((store) => {
        client.setStateStore(store);

        return client.createUser({
            username: 'peer'+userOrg+'Admin',
            mspid: getMspID(userOrg),
            cryptoContent: {
                privateKeyPEM: keyPEM,
                signedCertPEM: certPEM
            }
        });
    });
};
```

下面介绍创建通道的步骤：

- 1) 首先根据传入的 `channelConfigPath`，获取通道配置文件，提取为字节；
- 2) 把 `client` 实例切换到传入组织；

- 3) 使用传入组织的加密材料对通道配置字节签名;
- 4) 构建 request, 向 orderer 发送创建通道请求;
- 5) 创建成功则返回成功的结构对象, 失败则抛出异常。

```

var createChannel = function(channelName, channelConfigPath, username, orgName) {
    logger.debug('\n===== Creating Channel \'' + channelName + '\'' + '=====\n');
    var client = helper.getClientForOrg(orgName);
    var channel = helper.getChannelForOrg(orgName);

    // 取到通道配置文件
    var envelope = fs.readFileSync(path.join(__dirname, channelConfigPath));
    // 提取通道配置文件字节
    var channelConfig = client.extractChannelConfig(envelope);

    return helper.getOrgAdmin(orgName).then((admin) => {
        logger.debug(util.format('Successfully acquired admin user for
the organization "%s"', orgName));
        // 对通道配置字节签名为 "背书", 这是由 orderer 的通道创建策略所要求的
        let signature = client.signChannelConfig(channelConfig);

        let request = {
            config: channelConfig,
            signatures: [signature],
            name: channelName,
            orderer: channel.getOrderers()[0],
            txId: client.newTransactionID()
        };

        // 将创建通道请求发送给 orderer
        return client.createChannel(request);
    }, (err) => {
        logger.error('Failed to enroll user \'' + username + '\'. Error: ' +
err);
        throw new Error('Failed to enroll user \'' + username + '\'+ err);
    }).then((response) => {
        logger.debug(' response ::%j', response);
        if (response && response.status === 'SUCCESS') {
            logger.debug('Successfully created the channel.');
            let response = {
                success: true,
                message: 'Channel \'' + channelName + '\'' created
Successfully'
            };
            return response;
        } else {
            logger.error('\n!!!!!!!!!! Failed to create the channel \''
+ channelName +
            '\'+ !!!!!!!!!\n\n');

```

```

        throw new Error('Failed to create the channel \'' +
            channelName + '\');
    }
}, (err) => {
    logger.error('Failed to initialize the channel: ' + err.stack ?
        err.stack :
        err);
    throw new Error('Failed to initialize the channel: ' + err.stack ?
        err.stack : err);
});
};

exports.createChannel = createChannel;

```

## (2) 加入通道

加入通道包括如下步骤:

- 1) client 实例切换到传入组织;
- 2) 基于之前创建的通道, 获取该通道的创世区块;
- 3) 向要加入通道的 peers 发送加入通道的请求;

4) 在向 peers 发送加入通道请求的同时, 为各个 peer 分别注册 block eventhub 来监听区块产生的过程是否正常;

5) 通过校验第一个 peer 的 response status 是否为 200 来判断加入通道的结果, 成功则返回成功的结构对象, 失败则抛出异常。

```

var joinChannel = function(channelName, peers, username, org) {
    // 断开与 event hub 连接的函数
    var closeConnections = function(isSuccess) {
        if (isSuccess) {
            logger.debug('\n===== Join Channel is SUCCESS =====\n');
        } else {
            logger.debug('\n!!!!!!! ERROR: Join Channel FAILED !!!!!\n');
        }
        logger.debug('');
        for (var key in allEventhubs) {
            var eventhub = allEventhubs[key];
            if (eventhub && eventhub.isconnected()) {
                //logger.debug('Disconnecting the event hub');
                eventhub.disconnect();
            }
        }
    };

    //logger.debug('\n===== Join Channel =====\n');
    logger.info(util.format(
        'Calling peers in organization "%s" to join the channel', org));

    var client = helper.getClientForOrg(org);

```

```

var channel = helper.getChannelForOrg(org);
var eventhubs = [];

return helper.getOrgAdmin(org).then((admin) => {
    logger.info(util.format('received member object for admin of the
    organization "%s": ', org));
    tx_id = client.newTransactionID();
    let request = {
        txId: tx_id
    };
    return channel.getGenesisBlock(request);
}).then((genesis_block) => {
    tx_id = client.newTransactionID();
    var request = {
        targets: helper.newPeers(peers, org),
        txId: tx_id,
        block: genesis_block
    };

    eventhubs = helper.newEventHubs(peers, org);
    for (let key in eventhubs) {
        let eh = eventhubs[key];
        eh.connect();
        allEventhubs.push(eh);
    }

    var eventPromises = [];
    eventhubs.forEach((eh) => {
        let txPromise = new Promise((resolve, reject) => {
            let handle = setTimeout(reject, parseInt(config.
            eventWaitTime));
            eh.registerBlockEvent((block) => {
                clearTimeout(handle);
                if (block.data.length === 1) {
                    // 配置 block 只包括一个交易
                    var channel_header = block.data.
                    data[0].payload.header.channel_
                    header;
                    if (channel_header.channel_id
                    === channelName) {
                        resolve();
                    }
                    else {
                        reject();
                    }
                }
            });
        });
        eventPromises.push(txPromise);
    });
    return Promise.all(eventPromises);
});

```

// 一个 peer 可能属于多个通道，所以必须检查这个配置 block 是否来自于我们请求加入的通道

```

        eventPromises.push(txPromise);
    });
    let sendPromise = channel.joinChannel(request);
    return Promise.all([sendPromise].concat(eventPromises));
}, (err) => {
    logger.error('Failed to enroll user \'' + username + '\'' due to
    error: ' +
        err.stack ? err.stack : err);
    throw new Error('Failed to enroll user \'' + username +
        '\'' due to error: ' + err.stack ? err.stack : err);
}).then((results) => {
    logger.debug(util.format('Join Channel R E S P O N S E : %j',
    results));
    if (results[0] && results[0][0] && results[0][0].response &&
    results[0][0]
        .response.status == 200) {
        logger.info(util.format(
            'Successfully joined peers in organization %s to
            the channel \'' + s + '\'',
            org, channelName));
        closeConnections(true);
        let response = {
            success: true,
            message: util.format(
                'Successfully joined peers in organization
                %s to the channel \'' + s + '\'',
                org, channelName)
        };
        return response;
    } else {
        logger.error('Failed to join channel');
        closeConnections();
        throw new Error('Failed to join channel');
    }
}, (err) => {
    logger.error('Failed to join channel due to error: ' + err.stack ?
    err.stack :
        err);
    closeConnections();
    throw new Error('Failed to join channel due to error: ' + err.
    stack ? err.stack :
        err);
});
};
exports.joinChannel = joinChannel;

```

### (3) 安装链码

安装链码包括如下步骤:

- 1) client 实例切换到传入组织;
- 2) client 实例发出安装链码请求, 请求中包括目标 peers、链码路径、链码名称和链码



版本;

3) 对目标 peers 返回的 proposalResponses 结果依次校验, 所有 peers 都成功则返回成功的结构对象, 有 peer 失败则抛出异常。

```
var installChaincode = function(peers, chaincodeName, chaincodePath,
    chaincodeVersion, username, org) {
    logger.debug(
        '\n===== Install chaincode on organizations ====='
        '\n');
    helper.setupChaincodeDeploy();
    var channel = helper.getChannelForOrg(org);
    var client = helper.getClientForOrg(org);

    return helper.getOrgAdmin(org).then((user) => {
        var request = {
            targets: helper.newPeers(peers, org),
            chaincodePath: chaincodePath,
            chaincodeId: chaincodeName,
            chaincodeVersion: chaincodeVersion
        };
        return client.installChaincode(request);
    }, (err) => {
        logger.error('Failed to enroll user \'' + username + '\'. ' + err);
        throw new Error('Failed to enroll user \'' + username + '\'. ' + err);
    }).then((results) => {
        var proposalResponses = results[0];
        var proposal = results[1];
        var all_good = true;
        for (var i in proposalResponses) {
            let one_good = false;
            if (proposalResponses && proposalResponses[i].response &&
                proposalResponses[i].response.status === 200) {
                one_good = true;
                logger.info('install proposal was good');
            } else {
                logger.error('install proposal was bad');
            }
            all_good = all_good & one_good;
        }
        if (all_good) {
            logger.info(util.format(
                'Successfully sent install Proposal and received
                ProposalResponse: Status - %s',
                proposalResponses[0].response.status));
            logger.debug('\nSuccessfully Installed chaincode on
                organization ' + org +
                '\n');
            return 'Successfully Installed chaincode on organization '
                + org;
        } else {
            logger.error('Failed to enroll user \'' + username + '\'. ' + err);
        }
    });
}
```

```

        logger.error(
            'Failed to send install Proposal or receive valid
            response. Response null or status is not 200.
            exiting...'
        );
    };
    return 'Failed to send install Proposal or receive valid
    response. Response null or status is not 200. exiting...';
}
}, (err) => {
    logger.error('Failed to send install proposal due to error: ' +
        err.stack ?
            err.stack : err);
    throw new Error('Failed to send install proposal due to error: ' +
        err.stack ?
            err.stack : err);
});
});
exports.installChaincode = installChaincode;

```

#### (4) 实例化链码

实例化链码包括如下步骤：

- 1) client 实例切换到传入组织；
- 2) channel 调用 initialize(), 该方法会使用对应组织的 MSPs 实例化 channel 对象；
- 3) 发送背书 proposal 给 endorsers (args 里面指定的背书节点)；
- 4) 对目标 endorsers 返回的 proposalResponses 结果依次校验，所有 endorsers 都背书成功才进入下一步，有 endorsers 背书失败则抛出异常；
- 5) endorsers 背书成功后，应用端将背书 proposalResponses 和之前的 proposal 打包成 request，调用 sendTransaction 发给 orderer，这时因为 orderer 经过 order 后再通知 peers 进行实例化的操作是异步的，需要注册 transaction event 来监听实例化的最终结果；
- 6) 在 sendTransaction 和 transaction event 都成功返回的情况下，才说明实例化链码成功，此时返回成功的结构对象，若 transaction event 监听到失败则抛出异常。

```

var instantiateChaincode = function(channelName, chaincodeName, chaincodeVersion,
    functionName, args, username, org) {
    logger.debug('\n===== Instantiate chaincode on organization ' + org +
        ' =====\n');

    var channel = helper.getChannelForOrg(org);
    var client = helper.getClientForOrg(org);

    return helper.getOrgAdmin(org).then((user) => {

        // channel 实例从 orderer 读取该通道的配置区块，并基于所加入的组织实例化验证 MSPs
        return channel.initialize();
    }, (err) => {
        logger.error('Failed to enroll user \'' + username + '\'. ' + err);
    });
}

```

```

        throw new Error('Failed to enroll user \'' + username + '\'. ' + err);
    }).then((success) => {
        tx_id = client.newTransactionID();
        // 发送背书 proposal 给 endorser
        var request = {
            chaincodeId: chaincodeName,
            chaincodeVersion: chaincodeVersion,
            args: args,
            txId: tx_id
        };

        if (functionName)
            request.fcn = functionName;

        return channel.sendInstantiateProposal(request);
    }, (err) => {
        logger.error('Failed to initialize the channel');
        throw new Error('Failed to initialize the channel');
    }).then((results) => {
        var proposalResponses = results[0];
        var proposal = results[1];
        var all_good = true;
        for (var i in proposalResponses) {
            let one_good = false;
            if (proposalResponses && proposalResponses[i].response &&
                proposalResponses[i].response.status === 200) {
                one_good = true;
                logger.info('instantiate proposal was good');
            } else {
                logger.error('instantiate proposal was bad');
            }
            all_good = all_good & one_good;
        }
        if (all_good) {
            logger.info(util.format(
                'Successfully sent Proposal and received
                ProposalResponse: Status - %s, message - "%s",
                metadata - "%s", endorsement signature: %s',
                proposalResponses[0].response.status,
                proposalResponses[0].response.message,
                proposalResponses[0].response.payload,
                proposalResponses[0].endorsement
                .signature));
            var request = {
                proposalResponses: proposalResponses,
                proposal: proposal
            };

            // 设置一个 transaction listener 并且设置 30 秒 timeout
            // 如果在 timeout 的时限内, transaction 没有被有效提交则返回错误
            var deployId = tx_id.getTransactionID();

```

```

eh = client.newEventHub();
let data = fs.readFileSync(path.join(__dirname, ORGS[org].
peers['peer1']][
'tls_cacerts'
]));

eh.setPeerAddr(ORGS[org].peers['peer1']['events'], {
  pem: Buffer.from(data).toString(),
  'ssl-target-name-override': ORGS[org].peers['peer1']
  ['server-hostname']
});

eh.connect();

let txPromise = new Promise((resolve, reject) => {
  let handle = setTimeout(() => {
    eh.disconnect();
    reject();
  }, 30000);

  eh.registerTxEvent(deployId, (tx, code) => {
    logger.info(
      'The chaincode instantiate
      transaction has been committed
      on peer ' +
      eh._ep._endpoint.addr);
    clearTimeout(handle);
    eh.unregisterTxEvent(deployId);
    eh.disconnect();

    if (code !== 'VALID') {
      logger.error('The chaincode
      instantiate transaction was
      invalid, code = ' + code);
      reject();
    } else {
      logger.info('The chaincode
      instantiate transaction was valid.');
```

是 'sendTransaction()' 的调用结果

```

      resolve();
    }
  });
});

var sendPromise = channel.sendTransaction(request);
return Promise.all([sendPromise].concat([txPromise])).
then((results) => {
  logger.debug('Event promise all complete and
  testing complete');
  return results[0]; // Promise all 队列的第一个返回值
}).catch((err) => {

```

```

        logger.error(
            util.format('Failed to send instantiate
            transaction and get notifications within
            the timeout period. %s', err)
        );
        return 'Failed to send instantiate transaction and
        get notifications within the timeout period.';
    });
} else {
    logger.error(
        'Failed to send instantiate Proposal or receive
        valid response. Response null or status is not
        200. exiting...'
    );
    return 'Failed to send instantiate Proposal or receive
    valid response. Response null or status is not 200. exiting...';
}
}, (err) => {
    logger.error('Failed to send instantiate proposal due to error: '
    + err.stack ?
        err.stack : err);
    return 'Failed to send instantiate proposal due to error: ' +
    err.stack ?
        err.stack : err;
}).then((response) => {
    if (response.status === 'SUCCESS') {
        logger.info('Successfully sent transaction to the orderer.');
        return 'Chaincode Instantiation is SUCCESS';
    } else {
        logger.error('Failed to order the transaction. Error code:
        ' + response.status);
        return 'Failed to order the transaction. Error code: ' +
        response.status;
    }
}, (err) => {
    logger.error('Failed to send instantiate due to error: ' + err.
    stack ? err
        .stack : err);
    return 'Failed to send instantiate due to error: ' + err.stack ?
    err.stack :
        err;
});
};

exports.instantiateChaincode = instantiateChaincode;

```

### (5) 调用链码

调用链码和之前实例化链码步骤类似，也是需要先发出背书，包括如下步骤。

- 1) client 实例切换到传入组织。
- 2) 发送 proposal 给 endorsers (args 里面指定的背书节点)。

3) 对目标 endorsers 返回的 proposalResponses 结果依次校验, 所有 endorsers 都背书成功才进入下一步, 有 endorsers 背书失败则抛出异常。

4) endorsers 背书成功后, 应用端将背书 proposalResponses 和之前的 proposal 打包成 request, 调用 sendTransaction 发给 orderer, 这时因为 orderer 经过 order 后再通知 peers 进行调用链码的操作是异步的, 需要注册 transaction event 来监听调用链码的最终结果。

5) 在 sendTransaction 和 transaction event 都成功返回的情况下, 才说明调用链码成功, 此时返回成功的结构对象, 若 transaction event 监听到失败则抛出异常。

```
var invokeChaincode = function(peerNames, channelName, chaincodeName, fcn, args,
username, org) {
    logger.debug(util.format('\n===== invoke transaction on
organization %s =====\n', org));
    var client = helper.getClientForOrg(org);
    var channel = helper.getChannelForOrg(org);
    var targets = (peerNames) ? helper.newPeers(peerNames, org) : undefined;
    var tx_id = null;

    var txRequest = null;
    return helper.getRegisteredUsers(username, org).then((user) => {
        tx_id = client.newTransactionID();
        logger.debug(util.format('Sending transaction "%j"', tx_id));
        // 发送背书 proposal 给 endorser
        var request = {
            chaincodeId: chaincodeName,
            fcn: fcn,
            args: args,
            chainId: channelName,
            txId: tx_id
        };

        if (targets)
            request.targets = targets;
        var txRequest = channel.sendTransactionProposal(request)
            return txRequest;
    }, (err) => {
        logger.error('Failed to enroll user \'' + username + '\'. ' + err);
        throw new Error('Failed to enroll user \'' + username + '\'. ' + err);
    }).then((results) => {
        var proposalResponses = results[0];
        var proposal = results[1];
        var all_good = true;
        for (var i in proposalResponses) {
            let one_good = false;
            if (proposalResponses[i].response && proposalResponses[i].response.status === 200) {
                one_good = true;
                logger.info('transaction proposal was good');
            } else {
```

```

        logger.error(proposalResponses[i]);
        logger.error('transaction proposal was bad');
        if (proposalResponses[i].message != null) {
            return proposalResponses[i].message;
        }
        all_good = all_good & one_good;
    }
    if (all_good) {
        logger.debug(util.format(
            'Successfully sent Proposal and received
            ProposalResponse: Status - %s, message - "%s",
            metadata - "%s", endorsement signature: %s',
            proposalResponses[0].response.status,
            proposalResponses[0].response.message,
            proposalResponses[0].response.payload,
            proposalResponses[0].endorsement
            .signature));
        var request = {
            proposalResponses: proposalResponses,
            proposal: proposal
        };
        // 设置一个 transaction listener 并且设置 30 秒 timeout
        // 如果在 timeout 的时限内, transaction 没有被有效提交则返回错误
        var transactionID = tx_id.getTransactionID();
        var eventPromises = [];

        if (!peerNames) {
            peerNames = channel.getPeers().map(function(peer) {
                return peer.getName();
            });
        }

        var eventhubs = helper.newEventHubs(peerNames, org);
        for (let key in eventhubs) {
            let eh = eventhubs[key];
            eh.connect();

            let txPromise = new Promise((resolve, reject) => {
                let handle = setTimeout(() => {
                    eh.disconnect();
                    reject();
                }, 30000);

                eh.registerTxEvent(transactionID, (tx, code) => {
                    clearTimeout(handle);
                    eh.unregisterTxEvent(transactionID);
                    eh.disconnect();

                    if (code !== 'VALID') {

```



```

        logger.error(
            'The balance transfer
            transaction was invalid,
            code = ' + code);
        reject();
    } else {
        logger.info(
            'The balance transfer transaction
            has been committed on peer ' +
            eh._ep._endpoint.addr);
        resolve();
    }
    });
    });
    eventPromises.push(txPromise);
});
var sendPromise = channel.sendTransaction(request);
return Promise.all([sendPromise].concat(eventPromises)).
then((results) => {
    logger.debug(' event promise all complete and
    testing complete');
    return results[0]; // Promise all 队列的第一个返回值
    // 是 'sendTransaction()' 的调用结果
}).catch((err) => {
    logger.error(
        'Failed to send transaction and get
        notifications within the timeout period.'
    );
    return 'Failed to send transaction and get
    notifications within the timeout period.';
});
} else {
    logger.error(
        'Failed to send Proposal or receive valid response.
        Response null or status is not 200. exiting...'
    );
    return 'Failed to send Proposal or receive valid response.
    Response null or status is not 200. exiting...';
}
}, (err) => {
    logger.error('Failed to send proposal due to error: ' + err.
    stack ? err.stack :
        err);
    return 'Failed to send proposal due to error: ' + err.stack ?
    err.stack :
        err;
}).then((response) => {
    if (response.status === 'SUCCESS') {

```

```

        logger.info('Successfully sent transaction to the
orderer.');
```

```

        return tx_id.getTransactionID();
    } else {
        if (response.status != null) {
            logger.error('Failed to order the transaction. Error
code: ' + response.status);
            return 'Failed to order the transaction. Error code:
' + response.status;
        } else {
            return response;
        }
    }
}, (err) => {
    logger.error('Failed to send transaction due to error: ' + err.
stack ? err.stack : err);
    return 'Failed to send transaction due to error: ' + err.stack ?
err.stack : err;
});
};

exports.invokeChaincode = invokeChaincode;
```

## 12.4.2 链码功能实现

本节我们来看链码对外提供的功能接口和每个功能接口的实现过程。

### 1. 链码接口定义

链码接口由两部分组成，即调用函数名称和调用参数。

#### (1) 票据发布接口

票据发布的函数名称是 `issue`，只有一个参数，是 JSON 结构的 `Bill` 对象：

```

{
  "BillInfoID": "POC10000998",
  "BillInfoAmt": "222",
  "BillInfoType": "111",
  "BillInfoIsseDate": "20170910",
  "BillInfoDueDate": "20171112",
  "DrwrCmID": "111",
  "DrwrAcct": "111",
  "AccptrCmID": "111",
  "AccptrAcct": "111",
  "PyeeCmID": "111",
  "PyeeAcct": "111",
  "HodrCmID": "ACMID",
  "HodrAcct": "A 公司"
}
```

各字段参数说明如表 12-4 所示。

表 12-4 票据发布接口参数

参数名称	说明	参数名称	说明	参数名称	说明
BillInfoID	票据号码	DrwrCmID	出票人证件号码	PyeeAcct	收款人名称
BillInfoAmt	票据金额	DrwrAcct	出票人名称	HodrCmID	持票人证件号码
BillInfoType	票据类型	AccepTrCmID	承兑人证件号码	HodrAcct	持票人名称
BillInfoIsseDate	票据出票日期	AccepTrAcct	承兑人名称		
BillInfoDueDate	票据到期日期	PyeeCmID	收款人证件号码		

## （2）票据背书接口

票据背书的函数名称是 endorse，有 3 个参数按表 12-5 所示顺序。

表 12-5 票据背书接口参数

参数序号	参数说明	参数示例
1	票据号码	POC10000998
2	被背书人证件号码	ACMID
3	被背书人名称	A 公司

## （3）票据背书签收接口

票据背书签收的函数名称是 accept，有 2 个参数按表 12-6 的顺序。

表 12-6 票据背书签收接口参数

参数序号	参数说明	参数示例
1	票据号码	POC10000998
2	被背书人证件号码	BCMID
3	被背书人名称	B 公司

## （4）票据背书拒收接口

票据背书拒收的函数名称是 reject，有 2 个参数按表 12-7 的顺序。

表 12-7 票据背书拒收接口参数

参数序号	参数说明	参数示例
1	票据号码	POC10000998
2	被背书人证件号码	ACMID
3	被背书人名称	A 公司

## （5）查询持票人票据列表接口

查询持票人票据列表的函数名称是 queryMyBill，只有 1 个参数如表 12-8 所示。

表 12-8 查询持票人票据列表接口参数

参数序号	参数说明	参数示例
1	持票人证件号码	ACMID

## (6) 查询持票人待签收票据列表接口

查询持票人待签收票据列表的函数名称是 queryMyWaitBill，只有 1 个参数如表 12-9 所示。

表 12-9 查询持票人待签收票据列表接口参数

参数序号	参数说明	参数示例
1	持票人证件号码	ACMID

## (7) 根据票据号码查询票据信息接口

根据票据号码查询票据信息的函数名称是 queryByBillNo，只有 1 个参数如表 12-10 所示。

表 12-10 根据票据号码查询票据信息接口参数

参数序号	参数说明	参数示例
1	票据号码	POC10000998

## 2. 链码接口实现

链码初始化默认实现即可：

```
// chaincode Init 接口
func (a *BillChaincode) Init(stub shim.ChaincodeStubInterface) pb.Response {
    return shim.Success(nil)
}
```

链码调用接口包含票据发布、票据背书、票据签收、票据拒收、票据查询等：

```
// chaincode Invoke 接口
func (a *BillChaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
    function, args := stub.GetFunctionAndParameters()

    // invoke
    if function == "issue" {
        return a.issue(stub, args)
    } else if function == "endorse" {
        return a.endorse(stub, args)
    } else if function == "accept" {
        return a.accept(stub, args)
    } else if function == "reject" {
        return a.reject(stub, args)
    }

    // query
    if function == "queryMyBill" {
        return a.queryMyBill(stub, args)
    } else if function == "queryByBillNo" {
        return a.queryByBillNo(stub, args)
    } else if function == "queryMyWaitBill" {
        return a.queryMyWaitBill(stub, args)
    }
}
```

```

    res := getRetString(1,"ChainnovaChaincode Unkown method!")
    chaincodeLogger.Infof("%s",res)

    return shim.Error(res)
}

```

链码用到的一些通用接口:

// 链码返回结构

```

type chaincodeRet struct {
    Code int    // 0 成功 1 其他
    Des  string  // 描述
}

```

// 根据返回码和描述返回序列号后的字节数组

```

func getRetByte(code int,des string) []byte {
    var r chaincodeRet
    r.Code = code
    r.Des = des

    b,err := json.Marshal(r)

    if err!=nil {
        fmt.Println("marshal Ret failed")
        return nil
    }
    return b
}

```

// 根据返回码和描述返回序列号后的字符串

```

func getRetString(code int,des string) string {
    var r chaincodeRet
    r.Code = code
    r.Des = des

    b,err := json.Marshal(r)

    if err!=nil {
        fmt.Println("marshal Ret failed")
        return ""
    }
    chaincodeLogger.Infof("%s",string(b[:]))
    return string(b[:])
}

```

// 根据票号取出票据

```

func (a *BillChaincode) getBill(stub shim.ChaincodeStubInterface,bill_No string)
(Bill, bool) {
    var bill Bill
    key := Bill_Prefix + bill_No
    b,err := stub.GetState(key)

```

```

    if b==nil {
        return bill, false
    }
    err = json.Unmarshal(b,&bill)
    if err!=nil {
        return bill, false
    }
    return bill, true
}

// 保存票据
func (a *BillChaincode) putBill(stub shim.ChaincodeStubInterface, bill Bill) ([]byte, bool) {

    byte,err := json.Marshal(bill)
    if err!=nil {
        return nil, false
    }
    err = stub.PutState(Bill_Prefix + bill.BillInfoID, byte)
    if err!=nil {
        return nil, false
    }
    return byte, true
}

```

## (1) 票据发布

票据发布的实现如下:

```

// 票据发布
// args: 0 - {Bill Object}
func (a *BillChaincode) issue(stub shim.ChaincodeStubInterface, args []string) pb.Response {
    if len(args)!=1 {
        res := getRetString(1,"ChainnovaChaincode Invoke issue args!=1")
        return shim.Error(res)
    }
    var bill Bill
    err := json.Unmarshal([]byte(args[0]), &bill)
    if err!=nil {
        res := getRetString(1,"ChainnovaChaincode Invoke issue unmarshal failed")
        return shim.Error(res)
    }

    // 根据票号查找是否票号已存在
    _, existbl := a.getBill(stub, bill.BillInfoID)
    if existbl {
        res := getRetString(1,"ChainnovaChaincode Invoke issue failed :

```

```

        the billNo has exist ")
        return shim.Error(res)
    }

    if bill.BillInfoID == "" {
        bill.BillInfoID = fmt.Sprintf("%d", time.Now().UnixNano())
    }

    // 更改票据信息和状态并保存票据：票据状态设为新发布
    bill.State = BillInfo_State_NewPublish

    // 保存票据
    _, bl := a.putBill(stub, bill)
    if !bl {
        res := getRetString(1, "ChainnovaChaincode Invoke issue put bill
        failed")
        return shim.Error(res)
    }
    // 以持票人 ID 和票号构造复合 key 向 search 表中保存 value 为空即可 以便持票人批量查询
    holderNameBillNoIndexKey, err := stub.CreateCompositeKey(IndexName, []
    string{bill.HodrCmID, bill.BillInfoID})
    if err != nil {
        res := getRetString(1, "ChainnovaChaincode Invoke issue put search
        table failed")
        return shim.Error(res)
    }
    stub.PutState(holderNameBillNoIndexKey, []byte{0x00})

    res := getRetByte(0, "invoke issue success")
    return shim.Success(res)
}

```

## (2) 票据背书

票据背书的实现如下：

```

// 背书请求
// args: 0 - Bill_No ; 1 - Endorser CmId ; 2 - Endorser Acct
func (a *BillChaincode) endorse(stub shim.ChaincodeStubInterface, args []string)
pb.Response {
    if len(args) < 3 {
        res := getRetString(1, "ChainnovaChaincode Invoke endorse args<3")
        return shim.Error(res)
    }
    // 根据票号取得票据
    bill, bl := a.getBill(stub, args[0])
    if !bl {
        res := getRetString(1, "ChainnovaChaincode Invoke endorse get bill
        error")
        return shim.Error(res)
    }
}

```



```

if bill.HodrCmID == args[1] {
    res := getRetString(1, "ChainnovaChaincode Invoke endorse failed:
    Endorser should not be same with current Holder")
    return shim.Error(res)
}
// 更改票据信息和状态并保存票据：添加待背书人信息，重置已拒绝背书人，票据状态改为待背书
bill.WaitEndorserCmID = args[1]
bill.WaitEndorserAcct = args[2]
bill.RejectEndorserCmID = ""
bill.RejectEndorserAcct = ""
bill.State = BillInfo_State_EndrWaitSign

// 保存票据
_, bl = a.putBill(stub, bill)
if !bl {
    res := getRetString(1, "ChainnovaChaincode Invoke endorse put bill
    failed")
    return shim.Error(res)
}
// 以待背书人 ID 和票号构造复合 key 向 search 表中保存 value 为空即可 以便待背书人批量查询
holderNameBillNoIndexKey, err := stub.CreateCompositeKey(IndexName, []
string{bill.WaitEndorserCmID, bill.BillInfoID})
if err != nil {
    res := getRetString(1, "ChainnovaChaincode Invoke endorse put search
    table failed")
    return shim.Error(res)
}
stub.PutState(holderNameBillNoIndexKey, []byte{0x00})

res := getRetByte(0, "invoke endorse success")
return shim.Success(res)
}

```

### (3) 票据背书签收

票据背书签收的实现如下：

```

// 背书人接受背书
// args: 0 - Bill_No ; 1 - Endorser CmId ; 2 - Endorser Acct
func (a *BillChaincode) accept(stub shim.ChaincodeStubInterface, args []string)
pb.Response {
    if len(args) < 3 {
        res := getRetString(1, "ChainnovaChaincode Invoke accept args<3")
        return shim.Error(res)
    }
    // 根据票号取得票据
    bill, bl := a.getBill(stub, args[0])
    if !bl {
        res := getRetString(1, "ChainnovaChaincode Invoke accept get bill
        error")
        return shim.Error(res)
    }
}

```

// 维护 search 表：以前手持票人 ID 和票号构造复合 key 从 search 表中删除该 key 以便前手持票人无法再查到该票据

```
holderNameBillNoIndexKey, err := stub.CreateCompositeKey(IndexName, []
string{bill.HodrCmID, bill.BillInfoID})
if err != nil {
    res := getRetString(1, "ChainnovaChaincode Invoke accept put search
table failed")
    return shim.Error(res)
}
stub.DelState(holderNameBillNoIndexKey)
```

// 更改票据信息和状态并保存票据：将前手持票人改为背书人，重置待背书人，票据状态改为背书签收

```
bill.HodrCmID = args[1]
bill.HodrAcct = args[2]
bill.WaitEndorserCmID = ""
bill.WaitEndorserAcct = ""
bill.State = BillInfo_State_EndrSigned
```

// 保存票据

```
_, bl = a.putBill(stub, bill)
if !bl {
    res := getRetString(1, "ChainnovaChaincode Invoke accept put bill
failed")
    return shim.Error(res)
}

res := getRetByte(0, "invoke accept success")
return shim.Success(res)
}
```

#### (4) 票据背书拒收

票据背书拒收的实现如下：

// 背书人拒绝背书

// args: 0 - Bill\_No ; 1 - Endorser CmId ; 2 - Endorser Acct

```
func (a *BillChaincode) reject(stub shim.ChaincodeStubInterface, args []string)
pb.Response {
```

```
    if len(args)<3 {
        res := getRetString(1, "ChainnovaChaincode Invoke reject args<3")
        return shim.Error(res)
    }
```

// 根据票号取得票据

```
bill, bl := a.getBill(stub, args[0])
if !bl {
    res := getRetString(1, "ChainnovaChaincode Invoke reject get bill
error")
    return shim.Error(res)
}
```

```

// 维护 search 表：以当前背书人 ID 和票号构造复合 key 从 search 表中删除该 key 以便当
    前背书人无法再查到该票据
holderNameBillNoIndexKey, err := stub.CreateCompositeKey(IndexName, []
string{args[1], bill.BillInfoID})
if err != nil {
    res := getRetString(1, "ChainnovaChaincode Invoke reject put search
    table failed")
    return shim.Error(res)
}
stub.DelState(holderNameBillNoIndexKey)

// 更改票据信息和状态并保存票据：将拒绝背书人改为当前背书人，重置待背书人，票据状态改
    为背书拒绝
bill.WaitEndorserCmID = ""
bill.WaitEndorserAcct = ""
bill.RejectEndorserCmID = args[1]
bill.RejectEndorserAcct = args[2]
bill.State = BillInfo_State_EndrReject

// 保存票据
_, bl = a.putBill(stub, bill)
if !bl {
    res := getRetString(1, "ChainnovaChaincode Invoke reject put bill
    failed")
    return shim.Error(res)
}

res := getRetByte(0, "invoke accept success")
return shim.Success(res)
}

```

### (5) 票据信息查询

获取自己持有的票据的实现如下：

```

// 查询我的票据：根据持票人编号 批量查询票据
// 0 - Holder CmId ;
func (a *BillChaincode) queryMyBill(stub shim.ChaincodeStubInterface, args []
string) pb.Response {
    if len(args) != 1 {
        res := getRetString(1, "ChainnovaChaincode queryMyBill args!=1")
        return shim.Error(res)
    }
    // 以持票人 ID 从 search 表中批量查询所持有的票号
    billsIterator, err := stub.GetStateByPartialCompositeKey(IndexName, []
    string{args[0]})
    if err != nil {
        res := getRetString(1, "ChainnovaChaincode queryMyBill get bill list
        error")
        return shim.Error(res)
    }
}

```

```

defer billsIterator.Close()

var billList = []Bill{}

for billsIterator.HasNext() {
    kv, _ := billsIterator.Next()
    // 取得持票人名下的票号
    _, compositeKeyParts, err := stub.SplitCompositeKey(kv.Key)
    if err != nil {
        res := getRetString(1, "ChainnovaChaincode queryMyBill
        SplitCompositeKey error")
        return shim.Error(res)
    }
    // 根据票号取得票据
    bill, bl := a.getBill(stub, compositeKeyParts[1])
    if !bl {
        res := getRetString(1, "ChainnovaChaincode queryMyBill get
        bill error")
        return shim.Error(res)
    }
    billList = append(billList, bill)
}
// 取得并返回票据数组
b, err := json.Marshal(billList)
if err != nil {
    res := getRetString(1, "ChainnovaChaincode Marshal queryMyBill
    billList error")
    return shim.Error(res)
}
return shim.Success(b)
}

```

查询我的待背书票据实现如下:

```

// 查询我的待背书票据：根据背书人编号 批量查询票据
// 0 - Endorser CmId ;
func (a *BillChaincode) queryMyWaitBill(stub shim.ChaincodeStubInterface, args []
string) pb.Response {
    if len(args) != 1 {
        res := getRetString(1, "ChainnovaChaincode queryMyWaitBill args!=1")
        return shim.Error(res)
    }
    // 以背书人 ID 从 search 表中批量查询所持有的票号
    billsIterator, err := stub.GetStateByPartialCompositeKey(IndexName, []
string{args[0]})
    if err != nil {
        res := getRetString(1, "ChainnovaChaincode queryMyWaitBill
        GetStateByPartialCompositeKey error")
        return shim.Error(res)
    }
    defer billsIterator.Close()
}

```

```

var billList = []Bill{}

for billsIterator.HasNext() {
    kv, _ := billsIterator.Next()
    // 从 search 表中批量查询与背书人有关的票号
    _, compositeKeyParts, err := stub.SplitCompositeKey(kv.Key)
    if err != nil {
        res := getRetString(1, "ChainnovaChaincode queryMyWaitBill
        SplitCompositeKey error")
        return shim.Error(res)
    }
    // 根据票号取得票据
    bill, bl := a.getBill(stub, compositeKeyParts[1])
    if !bl {
        res := getRetString(1, "ChainnovaChaincode queryMyWaitBill
        get bill error")
        return shim.Error(res)
    }
    // 取得状态为待背书的票据 并且待背书人是当前背书人
    if bill.State == BillInfo_State_EndrWaitSign && bill.
    WaitEndorserCmID == args[0] {
        billList = append(billList, bill)
    }
}
// 取得并返回票据数组
b, err := json.Marshal(billList)
if err != nil {
    res := getRetString(1, "ChainnovaChaincode Marshal queryMyWaitBill
    billList error")
    return shim.Error(res)
}
return shim.Success(b)
}

```

根据票据号码查询票据的详细信息实现如下:

```

// 根据票号取得票据 以及该票据背书历史
// 0 - Bill_No ;
func (a *BillChaincode) queryByBillNo(stub shim.ChaincodeStubInterface, args []
string) pb.Response {
    if len(args)!=1 {
        res := getRetString(1, "ChainnovaChaincode queryByBillNo args!=1")
        return shim.Error(res)
    }
    // 取得该票据
    bill, bl := a.getBill(stub, args[0])
    if !bl {
        res := getRetString(1, "ChainnovaChaincode queryByBillNo get bill
        error")
    }
}

```

```

        return shim.Error(res)
    }

    // 取得背书历史：通过 fabric api 取得该票据的变更历史
    resultsIterator, err := stub.GetHistoryForKey(Bill_Prefix+args[0])
    if err != nil {
        res := getRetString(1,"ChainnovaChaincode queryByBillNo
        GetHistoryForKey error")
        return shim.Error(res)
    }
    defer resultsIterator.Close()

    var history []HistoryItem
    var hisBill Bill
    for resultsIterator.HasNext() {
        historyData, err := resultsIterator.Next()
        if err != nil {
            res := getRetString(1,"ChainnovaChaincode queryByBillNo
            resultsIterator.Next() error")
            return shim.Error(res)
        }

        var hisItem HistoryItem
        hisItem.TxId = historyData.TxId //copy transaction id over
        json.Unmarshal(historyData.Value, &hisBill)
        // un stringify it aka JSON.parse()
        if historyData.Value == nil { //bill has been deleted
            var emptyBill Bill
            hisItem.Bill = emptyBill //copy nil marble
        } else {
            json.Unmarshal(historyData.Value, &hisBill)
            // un stringify it aka JSON.parse()
            hisItem.Bill = hisBill //copy bill over
        }
        history = append(history, hisItem) //add this tx to the list
    }
    // 将背书历史作为票据的一个属性 一同返回
    bill.History = history

    b, err := json.Marshal(bill)
    if err != nil {
        res := getRetString(1,"ChainnovaChaincode Marshal queryByBillNo
        billList error")
        return shim.Error(res)
    }
    return shim.Success(b)
}

```

## 12.5 票据背书快速部署

在 Github 上提供了快速启动区块链网络和初始化的脚本。启动区块链网络和前端服务的脚本如下：

```
./setupFabricNetwork.sh &
```

创建通道及安装实例化链码的脚本如下：

```
./createChannelAndInstallChaincode.sh
```

上面的过程可能比较慢，等待出现“Total execution time”的日志就实例化结束了：

```
POST request Enroll on Org1 ...
{"success":true,"secret":"ILRegbALMUGw","message":"Jim enrolled Successfully","token":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE1MTIwODEwODcsInVzZXJuYWllIjoismltIiwib3JnTmFtZSI6Im9yZzEiLCJpYXQiOjE1MTIwNDU1ODd9.OeOZEaMiOH9vVc8LXapDGKV51lrhIoTb7PnLj9sriXYo"}
ZzEiLCJpYXQiOjE1MTIwNDU1ODd9.OeOZEaMiOH9vVc8LXapDGKV51lrhIoTb7PnLj9sriXYo}
ORG1 token is eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE1MTIwODEwODcsInVzZXJuYWllIjoismltIiwib3JnTmFtZSI6Im9yZzEiLCJpYXQiOjE1MTIwNDU1ODd9.OeOZEaMiOH9vVc8LXapDGKV51lrhIoTb7PnLj9sriXYo

POST request Enroll on Org2 ...
{"success":true,"secret":"cZNftXYCZBVJ","message":"Barry enrolled Successfully","token":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE1MTIwODEwODgsInVzZXJuYWllIjoiqmFycnkiLCJvcmd0YWllIjoib3JnMiIsImhhdCI6MTUxMjA0NTU0OH0.C7pWIUY7dNvmLIAj2j52GPWE5KsjwbBhzz8su5bZW0Y"}
ORG2 token is eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE1MTIwODEwODgsInVzZXJuYWllIjoiqmFycnkiLCJvcmd0YWllIjoib3JnMiIsImhhdCI6MTUxMjA0NTU0OH0.C7pWIUY7dNvmLIAj2j52GPWE5KsjwbBhzz8su5bZW0Y

POST request Create channel ...
{"success":true,"message":"Channel 'mychannel' created Successfully"}

POST request Join channel on Org1
{"success":true,"message":"Successfully joined peers in organization org1 to the channel 'mychannel'"}

POST request Join channel on Org2
{"success":true,"message":"Successfully joined peers in organization org2 to the channel 'mychannel'"}

POST Install chaincode on Org1
Successfully Installed chaincode on organization org1

POST Install chaincode on Org2
Successfully Installed chaincode on organization org2
```



```
POST instantiate chaincode on peer1 of Org1
Chaincode Instantiation is SUCCESS
```

```
Total execution time : 42 secs ...
```

接下来就可以通过 Web 页面访问系统。

12.6 票据背书展示

按照 `http://ip:4000/ng/src` 即可访问，我们下面看下实现的效果。

12.6.1 系统登录

系统登录页面如图 12-4 所示。

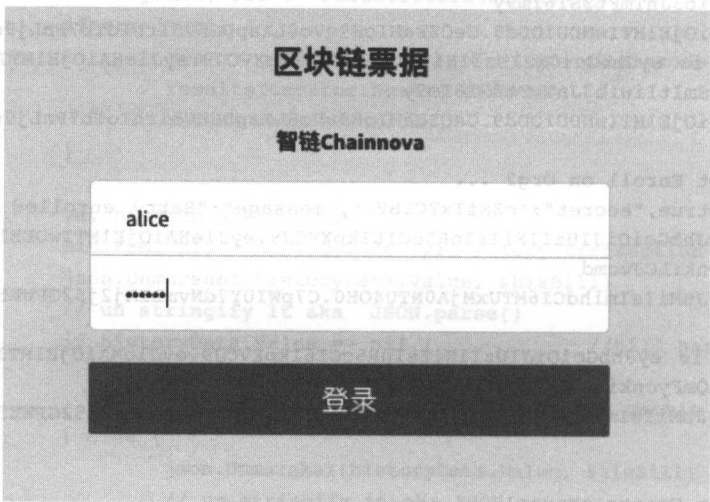


图 12-4 系统登录效果图

默认提供的用户名和密码如表 12-11 所示。

表 12-11 票据背书系统默认提供的用户名和密码

用户名	密码	名称	身份标识
admin	123456	管理员	HODR01
alice	123456	A 公司	ACMID
bob	123456	B 公司	BCMID
carle	123456	C 公司	CCMID

12.6.2 发布票据

发布票据页面如图 12-5 所示。

图 12-5 发布票据效果图

点击左边栏发布票据的选项，出现上图所示的发布票据页面，包含了票据的基本信息（票据号码、票据金额、票据类型、票据出票日期、票据到期日期等）、出票人信息（出票人名称、出票人证件号码等）、收款人信息（收款人名称、收款人证件号码等）、承兑人信息（承兑人名称、承兑人证件号码等）、持票人信息（持票人名称、持票人证件号码等）。票据号码是票据的唯一标识，是根据约定的规则离线生成的，发布票据的时候会自动监测是否存在重复票据。

### 12.6.3 我的票据

我的票据页面如图 12-6 所示。

票据号	票据状态	所属关系	操作
POC10000998	新发布	当前持有	详情

图 12-6 我的票据效果图

我的票据展示的是登录用户持有的所有票据，包含票号、票据状态、所属关系等基本信息，点击详情按钮会展示该票据的详细信息。

### 12.6.4 发起票据背书

发起票据背书页面如图 12-7 所示。

票据的详细信息除了发布票据时候录入的信息外，还展示了历史流转记录，包括交易号、操作业务、操作描述、当前持票人等信息。在浏览和确认了票据的详细信息后，可以选择发起票据背书，需要填入被背书人名称、被背书人证件号码等信息，就可以发起票据背书了。



图 12-7 发起票据背书效果图

12.6.5 待签收票据列表

待签收票据列表页面如图 12-8 所示。

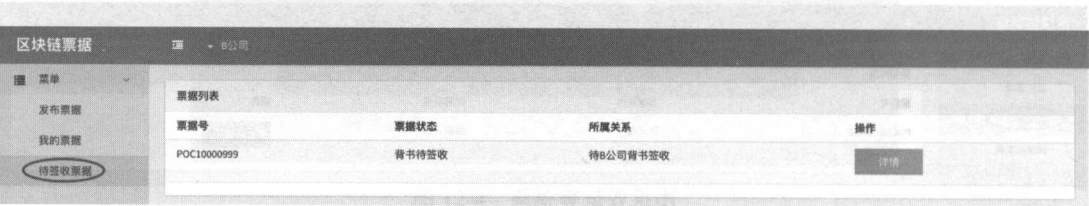


图 12-8 待签收票据列表效果图

被背书人登录区块链票据系统后，点击左边栏的待签收列表，右边能展示出所有的待签收票据列表。

12.6.6 签收票据背书

签收票据背书页面如图 12-9 所示。  
点击待签收票据的详情，会展示上图的票据信息。确认票据背书的内容后，可以选择签收背书或者拒绝背书。点击签收背书的按钮后，会提示签收成功，如图 12-10 所示。



图 12-9 签收票据背书效果图



图 12-10 票据背书已签收效果图

12.6.7 拒收票据背书

拒收票据背书页面如图 12-11 所示。



图 12-11 票据背书已拒绝效果图

拒收票据背书以后，票据的持票人还是原有的票据持票人，还可以继续发起新的票据背书，同时在票据的历史流转信息里也会记录票据背书和票据背书被拒收的信息。

## 12.7 本章小结

本章以一个票据背书的案例为背景，介绍如何基于 Hyperledger Fabric 1.0 开发区块链的应用程序。

程序开发的流程参考第 10 章的内容，先对票据背书的应用场景进行需求分析，对部分内容进行抽象和简化，设计了票据背书的分层架构和数据模型。实际的开发是基于 HFC Node.js SDK 开发后端服务，提供给前端的 Web 应用，业务逻辑的实现都是在链码实现的。本章只介绍了和区块链部分直接相关的后端服务和链码开发，前端的 Web 应用可以参考任何成熟的技术框架。

通过本章的例子，能够完整地了解如何基于 SDK 开发和区块链网络交互的应用程序，熟悉链码提供的主要接口及其功能，利用链码实现简单的业务逻辑。本章主要聚焦在 Hyperledger Fabric 1.0 相关的内容，示例的代码和实现并不是直接可用的，大家需要根据实际情况做优化和完善。



## 术 语 表

术语	英文	说明
多版本并发控制	MVCC (Multi-Version Concurrency Control)	MVCC 是保持通道中各 Peer 节点间状态同步的一种方法。Peer 节点并行地执行交易，在交易提交至账本之前，Peer 节点会检查交易在执行期间读到的数据是否被修改。如果读取的数据在执行和提交之间被改变，就会引发 MVCC 冲突，该交易会在账本中标记为无效，而且状态值不会更新到状态数据库中
崩溃故障容错	CFT (Crash Fault Tolerance)	在区块链网络中存在网络延时或者故障的情况下，共识机制能够确保有效的交易达成一致
拜占庭容错	BFT (Byzantine Fault Tolerance)	在区块链网络中存在部分恶意节点提交或者篡改请求的情况下，共识机制能够确保有效的交易达成一致
拜占庭将军问题	Byzantine failures	该问题是一个协议问题，指拜占庭帝国军队的将军们必须全体一致决定是否攻击某一支敌军。问题是这些将军在地理上是分隔开来的，并且将军中存在叛徒，而将军们只能依靠信使来传递信息。如何才能防止受到叛徒欺骗而做出错误决策呢？数学家设计的算法是让将军在接到其上一位将军标有进攻时间的信件之后，写上同意或反对并盖上自己的图章，然后把信转发给其他所有的将军，这样在信息周转之后，最后会出现一个盖有超过半数将军图章的信息链，以保证将军们在互不信任的情况下达成共识
背书	Endorsement	是指 Peer 节点调用智能合约模拟执行一个交易并对结果进行签名的过程
背书节点	Endorser	背书节点是 Peer 节点的子集，可对客户端提交的交易进行背书
背书策略	Endorsement Policy	对交易进行背书的规则，是与通道和链码相关的，在链码实例化的时候指定。链码调用的时候需要从背书节点收集到足够的签名背书，只有能够通过背书策略的交易才是有效的
创世区块	Genesis Block	创世区块是初始化区块链网络或通道的配置区块，也是链上的第一个区块
成员	Member	它是区块链网络中拥有独立根证书的合法独立实体。Peer 节点、排序服务节点和客户端都需要绑定到成员才能和网络中其他节点通信

(续)

术语	英文	说明
成员管理服务	Membership Services	成员管理服务是 Peer 节点、排序服务节点内置的服务,利用 PKI 体系颁发给成员的 X.509 证书,认证、授权和管理成员数字证书身份,提供授权的区块链操作
成员管理服务提供者	MSP (Membership Service Provider)	MSP 定义了成员管理服务的接口,目的是抽象化各成员之间的控制结构关系。MSP 在 X.509 数字证书的基础上,给每个成员绑定一个 MSP 标识,只有相同的 MSP 才能相互的认证
单进程的排序服务	Solo	排序服务节点只使用一个进程进行排序,进程内部使用 Golang 的协程和通道实现消息的排序和打包
读集	Read Set	包含唯一键的列表,还有在模拟执行过程中交易读取的已提交键值
读写集	RwSet (Read-Write Set)	包含读集和写集的集合
多链	Multi-Chain	区块链网络中的节点可以组成多个链,只有同一个链的 Peer 节点才能访问到链上的数据,这保护了用户数据的隐私
多通道	Multi-Channel	排序服务接收不同链上的数据进行排序打包以生成区块,排序服务节点采用多通道的方法来实现多链的数据隔离。每个链和通道一一对应,同一个链的数据提交到同一个通道进行处理
动态成员管理	Dynamic Membership	超级账本能够在不影响整个网络操作性的情况下,动态添加和移除成员、Peer 节点及排序服务节点。当业务关系调整或因各种原因需添加或移除实体时,动态成员管理就显得至关重要
反熵	Anti-entropy	每个节点周期性地和邻居节点交换保存的数据,然后对比本地数据和邻居节点的数据,检查是否有缺失或者过期的数据,然后更新本地节点的数据为最新的数据
Gossip 协议	Gossip Protocol	Gossip 数据传输协议有 3 项功能:1)管理 Peer 节点的发现和通道的成员关系;2)在同一个通道的所有 Peer 节点之间分发账本数据;3)在同一个通道的所有 Peer 节点之间同步状态数据
共识	Consensus	多个参与方对一个交易是否提交到账本以及提交的顺序达成一致的过程
共识安全性	Consensus Safety	指每个节点保证有相同的输入序列,并在每个节点上产生相同的输出结果
共识存活性	Consensus Liveness	是在没有通信故障的情况下,每个非故障节点最终都能接收到提交的所有交易
根 CA 证书	Root Certificate	自签名的证书,用根 CA 证书的私钥签名生成的证书还可以签发新的证书,形成一个树形结构
公钥基础设施	PKI (Public Key Infrastructure)	一种遵循标准并且利用公钥加密技术为电子商务的开展提供一套安全基础平台的技术和规范
工作量证明算法	PoW	依赖机器进行数学运算来获取记账权,即通过与或运算,计算出一个满足规则的随机数,从而获得本次记账权,发出本轮需要记录的数据,经全网其他节点验证后一起存储
股份授权证明	DPoS	类似于董事会投票,持币者投出一定数量的节点,代理他们进行验证和记账,持股人拥有所持股份对应的表决权
简化拜占庭容错算法	SBFT (Simplified BFT)	请参考: <a href="https://jira.hyperledger.org/browse/FAB-378">https://jira.hyperledger.org/browse/FAB-378</a>
记账	Commitment	通道中的每个 Peer 节点都会验证有序区块,并标记区块中每笔交易的状态是有效还是无效,然后将区块追加到本地账本中



(续)

术语	英文	说明
记账节点	Committer	所有的 Peer 节点都是记账节点, 负责验证排序服务节点打包到区块里的交易, 维护状态数据和账本的副本
交易	Transaction	交易是调用链码或实例化链码的结果, 它会提交给排序服务节点打包到区块中, 经过记账节点验证并保存到本地账本中。链码调用从账本读或者写数据的请求。链码实例化是在通道上启动和初始化链码的请求
交易背书 系统链码	ESCC (Endorsement System Chaincode)	主要功能是对交易结果进行结构转换和签名背书
交易验证 系统链码	VSCC (Validation System Chaincode)	主要功能是记账前对区块和交易进行验证
交易提案	Transaction Proposal	提交给背书节点的请求, 背书节点根据交易提案调用本地链码, 返回模拟执行的结果并背书签名
基于彩票 中奖的算法	Lottery-based Algorithm	基于彩票中奖的算法是一个形象化的说法, 加入到区块链网络中的节点都可以生成区块, 广播给网络中的其他节点进行确认, 通过竞争获得最终的记账权
基于投票 计数的算法	Voting-based Algorithm	基于投票计数的算法是节点参与区块的投票, 网络中的节点确认区块以后再记账的过程。在这种算法中区块的记账是确定性的, 已经记账的区块都是有效的
链	Chain	链是把交易打包到区块中, 区块按照时间顺序组成一种哈希链数据结构, 存储到文件系统中。Peer 节点从排序服务接收到区块, 根据背书策略和 MVCC 检查标记区块上的交易是否有效, 然后将该区块追加到 Peer 节点的本地账本中
链码	Chaincode	可独立运行的应用程序, 运行在基于 Docker 的安全容器中, 在启动的时候和背书节点建立 gRPC 连接, 在运行过程中通过链码接口和背书节点进行通信, 实现和账本的交互
链码安装	Chaincode Install	将链码放到 Peer 节点文件系统的过程
链码 部署交易	Chaincode Deploy Transaction	链码部署是执行链码实例化的交易, 是一种特殊的链码调用操作, 链码部署成功以后, 链码就部署到区块链上了
链码初始化	Chaincode Initialize	链码实例化或者升级的时候调用的初始化操作
链码查询	Chaincode Query	查询是一种对账本只读不写的链码调用, 可以读取一个或者多个键的状态值。由于查询不会更改账本状态, 所以客户端通常不会提交这些只读交易给排序服务打包到区块里
链码打包	Chaincode Package	生成包含链码源码、实例化策略和签名的文件, 基于这个文件进行链码实例化的操作
链码 调用交易	Chaincode Invoke Transaction	调用链码的操作, 实现智能合约的功能。只有链码部署和链码调用操作才能通过链码的接口修改账本的状态数据
链码实 例化	Chaincode Instantiate	在指定通道上启动并初始化链码的过程。初始化完成后, 安装过链码的 Peer 节点可以接收并处理链码调用请求
链码升级	Chaincode Upgrade	链码安装以后可以随时升级, 链码的名称需要保持不变, 必须要更新的是链码的版本, 其他部分, 比如链码的所有者和实例化策略都是可选的
历史数据	History	历史数据记录了每个状态数据的历史信息, 历史信息是保存在 LevelDB 数据库中的

(续)

术语	英文	说明
锚节点	Anchor Peer	锚节点是在通道配置中设置的, 一个组织一般会设置一个或者多个锚节点, 其他节点能够连接锚节点获取通道上存在的其他 Peer 节点
Peer 节点	Peer	Peer 节点是一个区块链网络实体, 它维护账本并运行链码容器来对账本执行读写操作
排序服务	Ordering Service	排序服务由多个排序服务节点组成, 这些节点将交易排序并打包成区块。排序服务是和 Peer 节点独立运行的, 按照时间顺序处理排序请求。排序服务是可插拔的设计方式, 目前默认实现了 Solo 和 Kafka 两种模式
排序服务节点	OSN (Ordering Service Node) 或者 Orderer	排序服务节点是接收交易进行排序并广播区块给 Peer 节点的节点。一般情况, 为了避免单点问题, 会部署多个排序服务节点
配置管理系统链码	CSCC (Configuration System Chaincode)	主要功能是管理记账节点上的配置信息
配置区块	Configuration Block	包含系统链或通道定义的成员和策略配置数据。对通道或整个网络的任何修改都会生成一个新的配置区块, 并追加到对应的链上。新的配置区块是海量配置项, 会包含创世区块的内容并加上最新的配置修改
区块	Block	区块是通道上包含有序交易的集合, 与上一个区块以密码学方式连接在一起
软件开发工具包	SDK (Software Development Kit)	SDK 的 API 提供交易处理、成员服务以及事件处理等功能, 用于编写和测试链码应用程序, 目前支持 Node.js、Golang、Java 和 Python
生命周期管理系统链码	LSCC (Lifecycle System Chaincode)	主要功能是管理部署在背书节点上的链码, 包括链码安装、实例化、升级等
索引数据	Index	对区块和区块中的交易建立索引, 并记录在索引数据库中, 能够快速通过交易号、区块号等方式查找区块和交易。目前只支持 LevelDB
提案	Proposal	发送给 Peer 节点的背书请求, 可以是链码实例化或者链码调用的请求
提案响应	Proposal Response	背书节点执行背书请求并签名的结果
通道	Channel	通道实现了数据的隔离和私密性, 加入到通道中的所有 Peer 节点之间共享同一个账本
写集	Write Set	包含了一个唯一键的列表和在模拟执行过程中交易写的键值
系统链	System Chain	系统链是一个特殊的链, 含有系统层面的全局配置区块链网络的联盟及组织信息、MSP 信息和策略信息等, 只存在于排序服务中
系统链码	System Chaincode	系统链码是一种特殊的链码, 运行在 Peer 节点的进程空间中。系统链码采用 Golang 的通道实现和 Peer 节点的通信, 减少了 gRPC 通信的开销。系统链码包含在 Peer 节点的可执行文件中, 不能通过普通链码的方式安装、实例化和升级
账本数据	Ledger	账本数据包括区块数据和状态数据, 每个 Peer 节点独立维护。区块数据是基于文件系统存储的, 每个链的账本数据存储在不同的目录下。状态数据记录在状态数据库中
主节点	Leading Peer 或者 Leader Peer	每一个成员在其订阅的通道上可以拥有多个 Peer 节点, 其中一个 Peer 节点会作为通道的主节点代表该通道与排序服务通信。排序服务将区块传递给主节点, 主节点再将此区块分发给同一成员下的其他 Peer 节点
状态数据	State	状态数据记录的是交易执行的结果, 最新的状态代表了通道上所有键的最新值, 所以又称为“世界状态”(World State)。链码不需要遍历每个区块就能快速地从状态数据库中读写数据, 目前支持 LevelDB 和 CouchDB

## 超级账本的实用工具

Hyperledger Fabric 1.0 提供了一些实用工具，方便和系统进行交互。

### B.1 协议转换工具 configtxlator

基于 Hyperledger Fabric 的区块链网络都是采用 gRPC 进行通信的，传输的消息和存储的区块采用的都是 Protocol Buffer 格式序列化的二进制结构，这不是对人友好的方式，尤其不方便修改。系统提供一个工具 configtxlator，它提供了 RESTful 接口的服务，其功能如下。

#### B.1.1 协议转换

可以在 Protocol Buffer 和 JSON 格式间直接转换，Protocol Buffer 还包含多种格式，有以下几种格式：

- ❑ common.Block：区块结构；
- ❑ common.Envelope：带有效载荷和数字签名的数字信封，区块的数据部分就是序列化后的数字信封；
- ❑ common.ConfigEnvelope：包含链配置的数字信封，内容包含 ConfigUpdateEnvelope；
- ❑ common.ConfigUpdateEnvelope：提交给排序节点的配置数字信封；
- ❑ common.Config：ConfigEnvelope 的配置部分；
- ❑ common.ConfigUpdate：ConfigUpdateEnvelope 的一部分。

工具 configtxlator 提供的是标准的 HTTP 服务，与任何可以处理 HTTP 协议的工具都

可以进行交互，下面用 CURL 工具来说明如何操作。

#### (1) Protocol Buffer 格式转换为 JSON 格式

```
curl -X POST --data-binary @config.pb
http://$SERVER:$PORT/protolator/decode/<message.Name> > config.json
```

其中，config.pb 是 Protocol Buffer 格式的文件，config.json 是解码后的 JSON 格式文件。  
\$SERVER 和 \$PORT 是 configtxlator 的服务和端口。<message.Name> 是消息的格式，可以为 common.Block 等。

#### (2) JSON 格式转换为 Protocol Buffer 格式

JSON 格式文件是纯文本格式的，可以用任何的文本编辑工具修改，也可以用专门的 JSON 处理工具 jq 快速地编辑文件，比如下面的命令把最大区块大小修改为 30 个交易：

```
jq ".channel_group.groups.Orderer.values.BatchSize.value.max_message_count = 30"
config.json > updated_config.json
```

编辑完成后再提交给 configtxlator，转换成对应的 Protocol Buffer 格式。

```
curl -X POST --data-binary @config.json http://$SERVER:$PORT/protolator/
encode/<message.Name> > config.pb
```

说明同上。

### B.1.2 配置更新计算

配置修改后，提交差异的部分给排序服务可以减少传输量。提交原始的配置文件和修改后的配置文件给通道，如下所示：

```
curl -X POST -F channel=desiredchannel -F original=@original_config.pb -F
updated=@updated_config.pb
http://$SERVER:$PORT/configtxlator/compute/update-from-configs > config_update.pb
```

其中，desiredchannel 是通道名称，original\_config.pb 和 updated\_config.pb 分别是原始的配置文件和修改后的配置文件，config\_update.pb 是生成的差异配置文件，它是 Protocol Buffer 格式的 common.ConfigUpdate 结构。

如果调用 SDK 更新配置，应用程序可以直接对这个差异配置文件签名，然后封装成 common.ConfigUpdateEnveloped，然后封装成 common.Envelope 提交给排序服务，结构的关系图如图 B-1 所示。

如果是命令行，在封装成 common.Envelope 之前需要先把 config\_update.pb 解码成 JSON 格式：

```
curl -X POST --data-binary @config_update.pb
http://$SERVER:$PORT/protolator/decode/common.ConfigUpdate > config_update.json
```

构造 common.Envelope 结构：

```
echo
'{"payload":{"header":{"channel_header":{"channel_id":"desiredchannel",
"type":2}}, "data":{"config_update":"'$(cat config_update.json)'}'}}' > config_update_
as_envelope.json
```

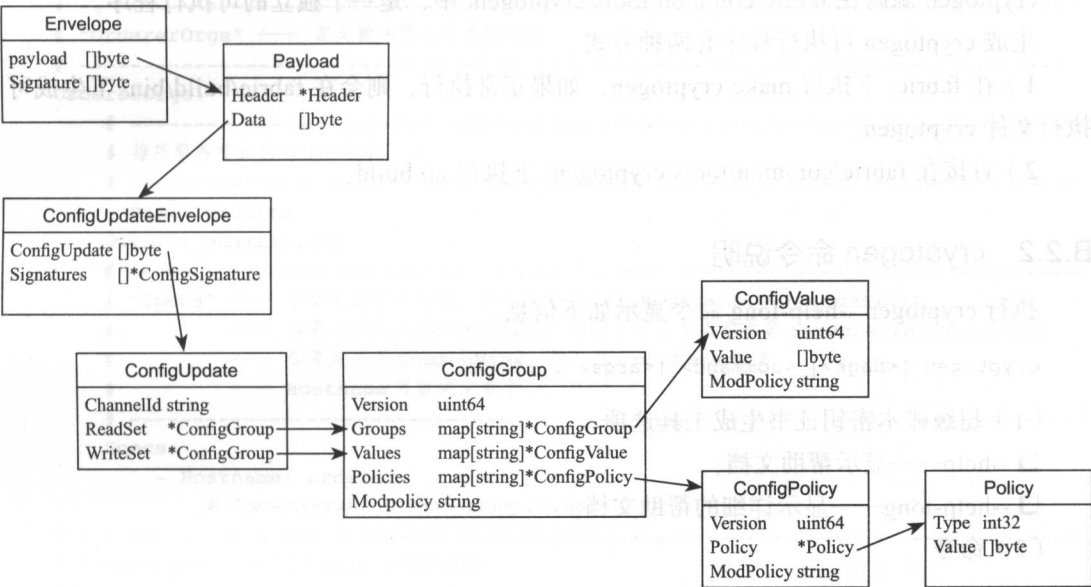


图 B-1

再转换成 Protocol Buffer 格式：

```
curl -X POST --data-binary @config_update_as_envelope.json
http://$SERVER:$PORT/protolator/encode/common.Envelope > config_update_as_
envelope.pb
```

调用命令行提交给排序节点：

```
peer channel update -f config_update_as_envelope.pb -c desiredchannel -o
$OSN_IP:$OSN_PORT
```

其中：\$OSN\_IP 和 \$OSN\_PORT 是排序服务节点的地址和端口。

B.2 MSP 生成工具 cryptogen

cryptogen 是 Hyperledger Fabric 提供的为网络实体生成加密材料（公钥、证书等）的实用程序。这些证书代表一个身份，并允许在网络实体间通信和交易时进行签名和身份认证。

cryptogen 使用一个包含网络拓扑的 crypto-config.yaml 文件，为文件中定义的组织 and 属于这些组织的实体生成一组证书和密钥。每个组织都配置了唯一的根证书（ca-cert），并包含了特定实体（peers 和 orders），这就形成了一种典型的网络结构——每个成员都有所属的

CA。Hyperledger Fabric 网络中的交易和通信都使用实体的私钥签名，使用公钥验证。

## B.2.1 编译生成 cryptogen 工具

cryptogen 源码在 fabric/common/tools/cryptogen/ 中，是一个独立的可执行程序。

生成 cryptogen 可执行程序有两种方式。

1) 在 fabric/ 下执行 make cryptogen，如果正常执行，则会在 fabric/build/bin/ 下生成可执行文件 cryptogen。

2) 直接在 fabric/common/tools/cryptogen/ 下执行 go build。

## B.2.2 cryptogen 命令说明

执行 cryptogen --help-long 命令显示如下信息。

```
cryptogen [<flags>] <command> [<args> ...]
```

(1) 超级账本密钥证书生成工具选项

❑ --help——显示帮助文档；

❑ --help-long——显示详细的帮助文档。

(2) 命令

```
help [<command>...]
```

1) 显示帮助文档。

```
generate [<flags>]
```

2) 生成密钥证书。

❑ --config——指定配置文件，如果不指定，则使用默认配置（即 cryptogen showtemplate 中的内容）；

❑ --output——生成密钥证书的目录，默认为 crypto-config。

3) showtemplate——显示配置模版。

4) version——显示版本信息。

最常用的命令语法是：

```
cryptogen generate --config=./crypto-config.yaml
```

根据 crypto-config.yaml 文件的配置，生成组织信息及其密钥证书等，保存在 crypto-config 目录下。

## B.2.3 crypto-config.yaml 文件解析

Crypto-config.yaml 是 cryptogen 工具使用的配置文件，cryptogen 工具根据该配置文件



生成加密材料。但该文件名字并非固定，也可自定义，只需在 `cryptogen generate` 命令中指定对应文件即可。

下面我们分析一下该文件的内容：

```
# -----
# "OrdererOrgs" —— 定义排序服务节点的组织
# -----
OrdererOrgs:
# -----
# 排序服务节点的名称和域名
# -----
- Name: Orderer
Domain: example.com
# -----
# "Specs" —— 手动定义节点名称，命名规范是 :{{.Hostname}}.{{.Domain}}
# —— 如果没有定义 CommonName，自动生成的节点名称是 :orderer.example.com
# —— 如果定义了 CommonName，就以 CommonName 为准
# —— Hostname 可以定义多个
# -----
Specs:
- Hostname: orderer
  # CommonName: order.example.net
# -----
# "PeerOrgs" —— 定义 Peer 节点的组织
# -----
PeerOrgs:
# -----
# 组织 Org1 的配置
# -----
- Name: Org1
  Domain: org1.example.com
# -----
# "CA" —— 默认会生成的名称 :ca.{{.Domain}}
# -----
# CA:
#   Hostname: ca
# -----
# "Template" —— 按模版生成 Peer 节点 MSP 证书
# —— 默认生成的 Peer 节点名称 :peer{{.Index}}.{{.Domain}}
# —— Index 是从 Start 到 Count-1，Start 默认是 0
# —— Hostname 可以自定义节点名称规则
# -----
Template:
  Count: 2
  # Start: 5
  # Hostname: {{.Prefix}}{{.Index}}
# -----
# "Users" —— 默认生成的用户数: User1@{{.Domain}}, User2@{{.Domain}}
```



```
# —— 默认生成 1 个管理员: Admin@{{.Domain}}
```

```
#
```

```
Users:
```

```
Count: 1
```

```
#
```

```
# 组织 Org2 的配置
```

```
#
```

```
- Name: Org2
```

```
Domain: org2.example.com
```

```
Template:
```

```
Count: 2
```

```
Users:
```

```
Count: 1
```

按照以上文件定义的内容,使用 `cryptogen generate` 命令生成的文件目录结构参考第 8 章的内容。

## B.3 配置生成工具 configtxgen

`configtxgen` 是 Hyperledger Fabric 提供的用于通道配置的实用程序,主要生成以下 3 种文件:

- ☐ 排序服务节点使用的创世区块;
- ☐ 创建通道使用的通道配置交易;
- ☐ 更新通道用的锚节点交易。

目前,该工具主要侧重于生成排序服务节点的创世区块,但是将来预计增加生成新通道的配置以及重新配置已有的通道。

### B.3.1 编译生成 configtxgen 工具

Cryptogen 源码在 `fabric/common/configtx/tool/configtxgen` 中,是一个独立的可执行程序。v1.0.0 之后的版本,源码转到 `fabric/common/tools/cryptogen/` 中。

生成 `cryptogen` 可执行程序有两种方式。

1) 在 `fabric` 目录下执行 `make configtxgen`。如果正常执行,则会在 `fabric/build/bin` 中生成可执行文件 `configtxgen`。

2) 直接在 `fabric/common/configtx/tool/configtxgen` 下执行 `go build`。

### B.3.2 configtxgen 命令说明

执行 `configtxgen --help` 命令显示如下信息。

(1) 用法

```
-asOrg string
```

## (2) 组织名称

- ❑ `-channelID string`——通道名称，默认是 "testchainid"；
- ❑ `-inspectBlock string`——检查和输出创世区块的内容；
- ❑ `-inspectChannelCreateTx string`——检查和输出通道创世区块的内容；
- ❑ `-outputAnchorPeersUpdate string`——创建锚节点配置，锚节点在 `configtx.yaml` 中的 `AnchorPeers` 中指定；
- ❑ `-outputBlock string`——创世区块生成的目录文件的名称，所用的 Profile 必须包含 `Consortiums`，否则启动排序服务节点会失败；
- ❑ `-outputCreateChannelTx string`——通道创世区块生成的目录文件的名称，所用的 Profile 必须包含 `Application`，否则创建通道会失败；
- ❑ `-profile string`——`configtx.yaml` 中的 Profiles 配置项，它描述了如何生成组织信息，默认的名称是 "SampleInsecureSolo"；
- ❑ `-version`——显示版本信息。

下面我们来看看命令 `configtxgen` 的常用用法。

### (1) 生成创世区块

生成创世区块的命令如下：

```
configtxgen -profile TwoOrgsOrdererGenesis
-outputBlock ./channel-artifacts/genesis.block
```

其中：

- ❑ `TwoOrgsOrdererGenesis` 为 `configtx.yaml` 中的 Profiles 之一；
- ❑ `./channel-artifacts/genesis.block` 为生成的创世区块文件名及保存路径。

生成的创世区块用于指定启动排序服务，还必须指定启动参数环境变量 `ORDERER_GENERAL_GENESISMETHOD=file` 和 `ORDERER_GENERAL_GENESISFILE=$PWD/genesis.block`，或者修改配置文件 `orderer.yaml` 编辑这些属性值。

### (2) 生成通道创世区块

生成通道创世区块的命令如下：

```
configtxgen -profile TwoOrgsChannel
-outputCreateChannelTx ./channel-artifacts/channel.tx -channelID
$CHANNEL_NAME
```

其中：

- ❑ `TwoOrgsChannel` 为 `configtx.yaml` 中的 Profiles 之一；
- ❑ `./channel-artifacts/channel.tx` 为生成的交易文件名及保存路径；
- ❑ `$CHANNEL_NAME` 为通道名称。

### (3) 生成组织锚节点

生成通道中的组织锚节点，如下所示：

```
configtxgen -profile TwoOrgsChannel
-outputAnchorPeersUpdate ./channel-artifacts/Org1MSPanchors.tx
-channelID CHANNEL_NAME -asOrg Org1MSP
```

其中：

- ❑ TwoOrgsChannel 为 configtx.yaml 中的 Profiles 之一；
- ❑ ./channel-artifacts/Org1MSPanchors.tx 为生成锚节点的配置文件名及保存路径；
- ❑ \$CHANNEL\_NAME 为通道名称；
- ❑ Org1MSP 为组织名称。

#### (4) 查看区块信息

查看生成的区块信息的示例如下所示：

```
configtxgen -profile TwoOrgsOrdererGenesis
-inspectBlock ./channel-artifacts/genesis.block
```

其中：./channel-artifacts/genesis.block 为指定的区块文件，该命令会将指定的区块文件解析成可读 JSON 格式信息展示出来。

#### (5) 查看通道配置信息

检查和查看通道创世区块的内容如下所示：

```
configtxgen -profile TwoOrgsOrdererGenesis
-inspectChannelCreateTx ./channel-artifacts/channel.tx
```

其中：./channel-artifacts/genesis.block 为指定的区块文件，该命令会将指定的区块文件解析成可读 JSON 格式信息展示出来。

### B.3.3 configtx.yaml 文件解析

configtxgen 工具的配置参数主要由 configtx.yaml 文件提供。在 fabric 库中，配置文件在 fabric/sampleconfig/configtx.yaml 中。这个配置文件可以编辑，或者通过设置环境变量来重写属性值，如 CONFIGTX\_ORDERER\_ORDERERTYPE=kafka。

此配置文件主要分为 3 部分。

1) Profiles 部分。它是默认的，这部分包含一些用于开发或测试场景的示例配置，这些配置涉及 fabric 目录中的加密部分。configtxgen 工具允许通过 -profile 标签来指定配置文件。Profiles 部分可以显式声明所有配置，但是通常都是从默认配置中继承。

2) Organizations 部分。它是默认的，这部分包含示例配置 MSP 定义的单一引用。对于生产部署，应该删除这部分配置，并以新网络成员的 MSP 定义来替代它。组织中每一个元素都必须带有锚标签，如 &orgName，这些标签可以在 Profiles 中部分引用。

3) 默认部分。此部分是 Orderer 和 Application 的配置，包括一些属性配置，如 BatchTimeout 和一般用作继承的基础值。

下面我们分析一下该文件的内容：

```
#####
# Profiles
#
# 可以编写不同的 profile 配置，作为参数给 configtxgen 工具使用
# 指定了 Consortium (组合、集团、联盟) profile 用来生成 orderer 的创世区块
# 带有正确联盟成员定义的 orderer 创世区块，其通道的创建请求必须带有组织成员名和联盟名
#
#####
Profiles:
  TwoOrgsOrdererGenesis:
    Orderer:
      <<: *OrdererDefaults
      OrdererType: kafka
      Organizations:
        - *OrdererOrg
    Consortiums:
      SampleConsortium:
        Organizations:
          - *Org1
          - *Org2
  TwoOrgsChannel:
    Consortium: SampleConsortium
    Application:
      <<: *ApplicationDefaults
      Organizations:
        - *Org1
        - *Org2

#####
# Section: Organizations
#
# - 该部分定义了这个配置文件中被引用的不同的组织标识
#
#####
Organizations:
  # SampleOrg 使用 sampleconfig 定义一个 MSP。这个 MSP 不会在生产中使用，但可以临时使用
  - &SampleOrg
    # DefaultOrg 定义一个在开发环境中 sampleconfig 使用过的组织
    Name: SampleOrg

    # load MSP 使用的 ID
    ID: DEFAULT

    # MSPDir 是 MSP 配置文件路径，由 cryptogen 工具生成的加密材料路径
    MSPDir: msp

    # AdminPrincipal 指定用于组织的管理员策略的主体类型
    # 目前只能使用 Role.ADMIN 和 Role.MEMBER，分别代表主体类型为 ADMIN 和 MEMBER
```

```

AdminPrincipal: Role.ADMIN

AnchorPeers:
    # AnchorPeers 定义了可用于跨组织 gossip 通信的 peer 的位置
    # 注意, 这个值只有在 Application 下使用时才会编码进创世区块 (即 profile 中 Application 下引用这个组织)
    - Host: 127.0.0.1
      Port: 7051

- &OrdererOrg
  Name: OrdererMSP
  ID: OrdererMSP
  MSPDir: crypto-config/ordererOrganizations/example.com/msp

- &Org1
  Name: Org1MSP
  ID: Org1MSP
  MSPDir: crypto-config/peerOrganizations/org1.example.com/msp
  AnchorPeers:
    - Host: peer0.org1.example.com
      Port: 7051

- &Org2
  Name: Org2MSP
  ID: Org2MSP
  MSPDir: crypto-config/peerOrganizations/org2.example.com/msp
  AnchorPeers:
    - Host: peer0.org2.example.com
      Port: 7051

#####
#
# SECTION: Orderer
#
# - 该部分定义了编码到配置交易或创世区块中的与 orderer 相关的参数值
#
#####
Orderer: &OrdererDefaults

    # Orderer Type: orderer 的启动方式
    # 可选类型是 "solo" 和 "kafka"
    OrdererType: solo

    Addresses:
        #- 127.0.0.1:7050
        - orderer.example.com:7050

    # Batch Timeout: 创建 batch 的超时时间
    BatchTimeout: 2s

```

# Batch Size: 控制 batch 到块中的消息数  
BatchSize:

# Max Message Count: batch 中允许的最大消息数量  
MaxMessageCount: 10

# Absolute Max Bytes: batch 中允许的绝对最大序列化消息字节数  
# 如果 OrdererType 是 kafka, 则要设置 Kafka brokers 的 'message.max.bytes' 和  
'replica.fetch.max.bytes' 的值大于该值  
AbsoluteMaxBytes: 10 MB

# Preferred Max Bytes: batch 中允许的首选绝对最大序列化消息字节数  
# 如果一个消息大于该值, 会导致 batch 大于该值  
PreferredMaxBytes: 512 KB

# Max Channels 是 order 网络中所允许的最大通道数  
# 设为 0 表示无最大通道数的限制  
MaxChannels: 0

Kafka:  
# Brokers: orderer 连接的 Kafka broker 列表  
# 注意: 使用 IP:port 格式  
Brokers:  
- 127.0.0.1:9092

# Organizations 是作为网络中 orderer 侧参与者的组织列表  
Organizations:

```
#####
#
# SECTION: Application
#
# - 该部分定义了编码到配置交易或创世区块中的与 APP 相关的参数值
# 注意, 在创建 channel tx 时才会用 Application, 否则是创建区块
#
#####
Application: &ApplicationDefaults
```

# Organizations 是作为网络中 APP 侧参与者的组织列表  
Organizations:

## 参考文献

- [1] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System[EB/OL]. <https://bitcoin.org/bitcoin.pdf>.
- [2] Gartner. Hype Cycle for Emerging Technologies[EB/OL]. <http://www.gartner.com/document/3383817>.
- [3] 梅兰妮·斯万. 区块链：新经济蓝图及导读 [M]. 北京：新星出版社，2016.
- [4] Wikipedia. Byzantine fault tolerance[EB/OL]. [https://en.wikipedia.org/wiki/Byzantine\\_fault\\_tolerance](https://en.wikipedia.org/wiki/Byzantine_fault_tolerance).
- [5] Hyperledger. 超级账本源代码 [EB/OL]. <https://github.com/hyperledger/fabric>.
- [6] Hyperledger. 超级账本源代码 [EB/OL]. <https://github.com/hyperledger/fabric-ca>.
- [7] Hyperledger. 超级账本源代码 [EB/OL]. <https://github.com/hyperledger/fabric-baseimage>.
- [8] Hyperledger. 超级账本源代码 [EB/OL]. <https://github.com/hyperledger/fabric-samples>.
- [9] Hyperledger. 超级账本项目管理 [EB/OL]. <https://jira.hyperledger.org>.
- [10] Hyperledger. 超级账本文档 [EB/OL]. <http://hyperledger-fabric.readthedocs.io>.
- [11] 智库. 票据 [EB/OL]. <http://wiki.mbalib.com/wiki/票据>.
- [12] 亿欧. 以京东金融为例解析区块链数字票据 [EB/OL]. <http://www.iyiou.com/p/36203>.



## 内容简介

超级账本执行董事 Brian Behlendorf 领衔推荐，资深一线区块链专家联合撰写，区块链和 Hyperledger 技术扛鼎之作。本书深度剖析区块链框架 Hyperledger Fabric 1.0 的架构、核心技术、部署与应用开发。

### 本书分三篇，共 12 章。

**准备篇（第 1 ~ 2 章）**，言简意赅地介绍区块链的核心理念、演进、主流平台、商用场景，以及安装、部署与调试。这两章为读者厘清思路，并可初步直观感受区块链。

**核心篇（第 3 ~ 9 章）**，庖丁解牛式剖析 Fabric 1.0 架构到内部实现机制，该篇是本书的核心，是理解区块链设计与实现的关键所在，也是日后应用的基础。该篇涵盖：

- Fabric 架构、组件关系与运行机制总览；
- Gossip 协议与 P2P 数据分发机制；
- 分布式账本数据相关的存储技术；
- 共识机制及其可插拔架构设计；
- 如何实现数据隔离的多链与多通道；
- 基于数字证书的成员管理服务实现与使用；
- 智能合约实现、交互，以及有限状态机。

**应用篇（第 10 ~ 12 章）**，从安装部署、开发模型和应用开发的角度，结合票据背书案例讲解如何基于 Hyperledger Fabric 1.0 开发区块链应用，以完整地掌握区块链应用开发，动手实践具体的项目。

写作投稿：165075460@qq.com



本书集合了多方共同的心血，由 Linux 基金会会员、智链 ChainNova 的一线技术团队主笔撰写，他们不仅是超级账本中国社区的主要贡献者，还长期奋战在市场前线，对商业应用环境有相当的了解，相信从书中内容的翔实程度可见一斑。

—— **Brian Behlendorf** 超级账本执行董事

这是我迄今为止所见关于超级账本技术和应用非常有参考意义的技术书籍，值得向广大区块链的研究者与开发同行们推荐。

—— **陈钟** 北京大学信息科学技术学院教授、北京大学金融信息化研究中心主任

本书的目的不是蜻蜓点水地介绍一些 Hyperledger 入门知识，而是通过阅读本书能让读者达到一定的水平，甚至可以加入区块链产业应用中来，为区块链的发展和实践落地添砖加瓦。

—— **何宝宏** 中国信息通信研究院云计算与大数据研究所所长

本书作者均来自超级账本会员企业骨干团队，拥有多年一线实践经验，这让本书不仅内容翔实，更具备很强的可操作性。作为中国技术工作组的核心成员，智链的成员为超级账本项目的国际化和技术推广都做出了重要贡献。无论是想了解先进的企业级区块链技术，还是进行应用实践参考，本书都值得一读。

—— **杨保华** 超级账本全球技术委员会委员，中国技术工作组主席

董宁牵头编写的《深度探索区块链：Hyperledger 技术与应用》系统阐述了超级账本的技术原理、架构、核心组件及应用开发实例，是一本学习超级账本底层技术和应用实例的优秀工具书，值得推荐。

—— **马小峰** 苏州同济金融科技研究院院长、中国电子学会区块链专委会副主任委员

可以预见，不久的将来区块将成为数字社会的基础设施。作为富含 IBM 基因的超级账本资深参与者，作者对超级账本有着深刻的理解。本书从区块链的概述到超级账本的技术解读与操作，处处见功夫，是非常有价值的读本。

—— **孙贻滋** 中国电子学会区块链专委会副主任委员兼秘书长

**CSDN** **Geekbang** 联合推荐  
极客邦科技



投稿热线：(010) 88379604  
客服热线：(010) 88379426 88361066  
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com  
网上购书：www.china-pub.com  
数字阅读：www.hzmedia.com.cn

